5-1-1998

Coarse-grained parallel genetic algorithms: Three implementations and their analysis

Daniel Pedersen

Coarse-Grained Parallel Genetic Algorithms

Three Implementations and Their Analysis

by

Daniel R. Pedersen

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Dr. Mohammad E. Shaaban, Assistant Professor

Dr. Peter G. Anderson, Professor of Computer Science

Dr. Roy. S. Czernikowski, Professor and Department Head

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
May 1998

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY

COLLEGE OF ENGINEERING

Title: Parallel Genetic Algorithms: Three Implementations and Their Analysis

I, Daniel R. Pedersen, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or in part.

5/18/93

Daniel R. Pedersen

Parallel Genetic Algorithms

Three Implementations and Their Analysis

by

Daniel R. Pedersen

Acknowledgment

I would like to thank my wonderful wife, Jenine, and my children, Erika and Jaime, for all of their love, support and understanding during this year I have been "away." I would also like to thank Charlie Datz and the UBT/TDT development team of Harris RF Communications for their patience and PCs.

Abstract

Although solutions to many problems can be found using direct analytical methods such as those calculus provides, many problems simply are too large or too difficult to solve using traditional techniques. Genetic algorithms provide an *indirect* approach to solving those problems. A genetic algorithm applies biological genetic procedures and principles to a randomly generated collection of potential solutions. The result is the evolution of new and better solutions. Coarse-Grained Parallel Genetic Algorithms extend the basic genetic algorithm by introducing genetic isolation and distribution of the problem domain.

This thesis compares the capabilities of a serial genetic algorithm and three coarse-grained parallel genetic algorithms (a standard parallel algorithm, a non-uniform parallel algorithm and an adaptive parallel algorithm). The evaluation is done using an instance of the traveling salesman problem. It is shown that while the standard course-grained parallel algorithm provides more consistent results than the serial genetic algorithm, the adaptive distributed algorithm out-performs them both. To facilitate this analysis, an extensible object-oriented library for genetic algorithms, encompassing both serial and coarse-grained parallel genetic algorithms, was developed. The Java programming language was used throughout.

Table of Contents

List of Figures ____ *viii*

List of Tables ____ *x*

Glossary ____ *xi*

1. Introduction ____ *1*

2. Background and Theory ____ *3*

2.1 Genetic Algorithms ____ **3**

2.1.1 Fitness Evaluation Functions ____ 4

2.1.2 Termination Criteria ____ 4

2.1.3 Problem Representation ____ 5

2.1.4 Schema Theory ____ 6

2.1.5 Initialization and Population Sizing ____ 6

2.1.6 Selection Methods ____ 7

2.1.6.1 Fitness Proportional Selection ____ 7

2.1.6.2 Tournament Selection ____ 9

2.1.7 Crossover Methods ____ 9

2.1.7.1 Bit Crossover Methods ____ 10

2.1.7.2 Permutation-Based Crossover Methods ____ 12

2.1.8 Mutation ____ 15

2.1.8.1 Bit Chromosome Mutation ____ 16

2.1.8.2 Permutation Chromosome Mutation ____ 16

2.1.9 Replacement Policies ____ 17

2.1.9.1 Generational ____ 17

2.1.9.2 Steady-State ____ 17

2.1.10 Premature Convergence ____ 17

2.2 Parallel Genetic Algorithms ____ **19**

2.2.1 Global Parallel ____ 19

2.2.2 Coarse Grained Parallel ____ 20

2.2.3 Fine Grained Parallel ____ 22

2.2.4 Hybrids ____ 23

2.2.5 Non-Uniform Distributed ____ 24

2.2.6 Adaptive Distributed ____ 24

2.2.7 Island-Injection ____ 25

3. Implementation Overview ____ *27*

3.1 Java ____ **27**

3.1.1 Simplicity ____ 27

3.1.2 Portable ____ 28

3.2 Remote Method Invocation ____ **30**

3.2.1.1 The Mechanics ____ 30

3.3 Design Patterns 32
3.3.1 Creational Patterns 32
3.3.1.1 Factory Method 32
3.3.1.2 Singleton 33
3.3.2 Structural Patterns 34
3.3.2.1 Adapter 34
3.3.2.2 Proxy 36
3.3.3 Behavioral Patterns 37
3.3.3.1 Observer 37
3.3.3.2 Strategy 38
3.4 TSPLIB 95 39
3.5 Primary Packages 39
3.5.1 Basic GA package 40
3.5.1.1 Chromosome Classes 40
3.5.1.2 GA Utility Classes 41
3.5.1.3 GA Resource Classes 43
3.5.1.4 Selection Method Classes 43
3.5.1.5 Crossover Method Classes 44
3.5.1.6 GA Style Classes 44
3.5.1.7 GA Package Summary 46
3.5.2 Basic CGGA package 47
3.5.2.1 Support Objects 47
3.5.2.2 Deme Objects 48
3.5.2.3 Master Objects 49
3.5.2.4 CGGA Package Summary 51
3.5.3 GA2 Package 51
3.5.3.1 Supervisory GA Classes 51
3.5.3.2 GA2 Deme Classes 51
3.5.3.3 GA2 Master Classes 52
3.5.3.4 GA2 Package Summary 52
3.5.4 Thesis Package 52
3.5.4.1 Thesis Utilities 52
3.5.4.2 Thesis Serial GA 53
3.5.4.3 Thesis Master Classes 53
3.5.4.4 Thesis Deme Classes 54
3.6 Supporting Packages 55
3.6.1 awt 55
3.6.2 lang 55
3.6.3 io 56
3.6.4 math 58
3.6.5 net 59
4. Data and Analysis 61
4.1 Basic Serial Genetic Algorithm 62

4.2 Basic Coarse-Grained Genetic Algorithm 65
4.3 Non-Uniform Distributed Genetic Algorithm 67
4.4 Adaptive Distributed Genetic Algorithm 68
4.5 Side-By-Side Comparison 75
5. Conclusions *77*
5.1 Future Code Development 77
5.2 Future Research Areas 78
6. Bibliography *79*

List of Figures

FIGURE 1: THE BASIC GA SEQUENCE 3
FIGURE 2: BASIC ROULETTE WHEEL SELECTION ALGORITHM 8
FIGURE 3: RWS WITH RANK SCALING EQUATION 8
FIGURE 4: RWS WITH SIGMA SCALING EQUATION 9
FIGURE 5: SINGLE POINT CROSSOVER EXAMPLE 10
FIGURE 6: DOUBLE POINT CROSSOVER EXAMPLE 11
FIGURE 7: UNIFORM CROSSOVER EXAMPLE 11
FIGURE 8: CYCLE CROSSOVER EXAMPLE 13
FIGURE 9: ORDERED CROSSOVER EXAMPLE 14
FIGURE 10: PARTIAL MATCHED CROSSOVER EXAMPLE 15
FIGURE 11: EXAMPLE GLOBAL PARALLEL GA 19
FIGURE 12: EXAMPLE COARSE-GRAINED PARALLEL GA 20
FIGURE 13: EXAMPLE FINE-GRAINED PARALLEL GA 23
FIGURE 14: A GOOD GA2 FITNESS FUNCTION 25
FIGURE 15: EXAMPLE FACTORY METHOD PATTERN BOOCH DIAGRAM 33
FIGURE 16: EXAMPLE SINGLETON PATTERN BOOCH DIAGRAM 34
FIGURE 17: EXAMPLE CLASS ADAPTER PATTERN BOOCH DIAGRAM 35
FIGURE 18: EXAMPLE OBJECT ADAPTER PATTERN BOOCH DIAGRAM 35
FIGURE 19: EXAMPLE PROXY PATTERN BOOCH DIAGRAM 36
FIGURE 20: EXAMPLE OBSERVER PATTERN BOOCH DIAGRAM 37
FIGURE 21: EXAMPLE STRATEGY PATTERN BOOCH DIAGRAM 38
FIGURE 22: SAMPLE INI FILE 57
FIGURE 23: SAMPLE TSP DATA FILE (ATT48.TSP) 58
FIGURE 24: ATT48 WITH TSPLIB OPTIMAL TOUR 61
FIGURE 25: SERIAL GA SCORES 63
FIGURE 26: ATT48 SERIAL GA SEED 100109 TOUR 64
FIGURE 27: SERIAL AND CGGA SCORES 66
FIGURE 28: SERIAL AND NUDGA SCORES 68
FIGURE 29: ATT48 GA2 SEED 100049 TOUR 69

FIGURE 30: GA2 TOURNAMENT SIZES FOR SEED 100049 70
FIGURE 31: GA2 POPULATION SIZES FOR SEED 100049 71
FIGURE 32: GA2 MUTATION RATES FOR SEED 100049 72
FIGURE 33: GA2 MIGRATION RATES FOR SEED 100049 73
FIGURE 34: GA2 CROSSOVER METHODS FOR SEED 100049 74
FIGURE 35: SERIAL AND GA2 SCORES 75
FIGURE 36: ALL GA SCORES 76

List of Tables

TABLE 1: KEY SERIAL GA STATISTICS ____ 62
TABLE 2: KEY CGGA STATISTICS ____ 65
TABLE 3: KEY NUDGA STATISTICS ____ 67
TABLE 4: BEST NUDGA PARAMETERS ____ 67
TABLE 5: KEY GA2 STATISTICS ____ 69
TABLE 6: ALL GA STATISTICS SIDE-BY-SIDE ____ 76

Glossary

Note: All significant terms first appear as underlined text in the main body of this thesis.

Term	Page	Definition
ATSP	1	Asymmetric Traveling Salesman Problem
CGGA	20	Coarse-Grained Genetic Algorithm
deme	20	A node in a CGGA
EA	3	Evolutionary Algorithm
Elitism	17	A technique for forcibly retaining individuals within a population
FGGA	22	Fine-Grained parallel Genetic Algorithm
Fitness Sharing	18	A technique for reducing premature convergence
GA	3	Genetic Algorithm
GA2	24	Adaptive distributed Genetic Algorithm
GARAGe	23	Genetic Algorithm Research and Applications Group, Michigan State University
iiGA	25	Island-Injection Genetic Algorithm
JVM	29	Java Virtual Machine
L	5	Chromosome length
NP-Complete	1	Nondeterministic Polynomial Complete
NUDGA	24	Non-Uniform Distributed Genetic Algorithm.
Population Crowding	18	A technique for reducing premature convergence
RMI	30	Remote Method Invocation
RPC	30	Remote Procedure Call
RWS	7	Roulette Wheel Selection
Schema or Schemata	6	Useful information encoding blocks for analyzing genetic operators
STSP	1	Symmetric Traveling Salesman Problem
TCP/IP	32	Transfer Control Protocol/Internet Protocol
TSP	1	Traveling Salesman Problem
URL	77	Uniform Resource Locator
VLSI	1	Very Large-Scale Integration

1. Introduction

Of significance to the computer engineering community are the optimal routing problems related to the design and layout of Very Large-Scale Integrated Circuits (VLSI). Designs with short traces and few vias result in layouts that are more reliable, lower cost and easier to maintain or rework. While not a VLSI layout problem, Traveling Salesman Problems (TSPs) are a class of similar problems that are NP-Complete. Algorithms able to solve a TSP can be adapted for VLSI layout problems. This thesis specifically works with Symmetric Traveling Salesman Problems (STSP). These are TSP problems where the distance (or cost) between two cities is the same when traveled from city 1 to city 2 as when traveled from city 2 to city 1. An Asymmetric TSP (ATSP) is a problem where the distance (or cost) between two cities is **not** the same.

A genetic algorithm is a technique for achieving a problem's solution through an indirect means. Randomly generated information that encodes potential solutions is manipulated by software models of natural genetics and evolution. The outcome is the generation of encoded solutions that are continually improved. Genetic algorithms are often placed in the role of optimizer for problems that are too difficult for direct analytical devices. However, they have also been successfully applied in many other areas including classifier systems (the abstraction of complementary rule collections) and the development of antibodies to antigens [SFP93, p. 145].

Coarse-grained parallel genetic algorithms are extensions of the basic serial algorithm. Multiple instances of the serial algorithm are executed concurrently with periodic communication between the instances to exchange best know sets of genetic material. The instances are effectively isolated for a period of time allowing independent development of their own genetic material. The information exchange updates each instance with new and possibly different material that may allow the instance to more quickly achieve the desired goal. The multiple instances of the coarse-grain algorithm

allows more of the fitness landscape to be examined at one time as compared to the serial algorithm.

This paper compares the ability of four genetic algorithm implementations to solve an instance of the Traveling Salesman Problem, ATT48 from the TSPLIB archive [TSPLIB]. The four implementations are: a serial genetic algorithm, a basic coarse-grained parallel genetic algorithm, a non-uniform distributed genetic algorithm and an adaptive distributed genetic algorithm. The later two implementations are advanced derivatives of the basic coarse-grained algorithm. It is shown that while the standard course-grained parallel algorithm provides more consistent results than the serial genetic algorithm, the adaptive distributed algorithm can find better solutions even more consistently.

Coarse-grained parallel genetic algorithms can be implemented as a distributed system. The Java programming language is a simple, interpreted, object-oriented language developed by Sun Microsystems that is currently popular for the development of distributed systems. This thesis utilized the Java programming language, and its Remote Method Invocation features, to develop an object-oriented, reusable, and extensible distributed coarse-grained parallel genetic algorithms library.

Section 2 of this thesis presents a significant discussion of the essential background points to the development of this thesis: genetic algorithms and parallel genetic algorithm. Section 3 introduces the reader to Java, Remote Method Invocation, and object-oriented software design patterns that were useful to the development of the thesis' code. Additionally, a discussion and overview of the design and decisions related to the implementation is supplied. In section 4, the results of the experiments on the developed code are presented along with an analysis of those results. Finally, section 5 gives some concluding remarks about the thesis and the results followed by an enumeration of what can be examined next.

2. Background and Theory

2.1 Genetic Algorithms

Initial research into evolutionary strategies involved solving only specific problems without consideration as to how these strategies actually worked. After a decade of development, John Holland officially introduced genetic algorithms in 1975 as "an abstraction of biological evolution." [MM96, pp. 2, 3]. It was his intent to create a formal method by which to study evolutionary strategies. The general Evolutionary Algorithm (EA) uses asexual reproduction [MH90, p. 2]. For example, Simulated Annealing (SA) implementations typically execute using a single solution (a population size of one) which is mutated into another individual who is scored using some measurement function. The new individual is accepted to replace the original individual based upon some predefined probability function. The probability function typically varies with the length of time of the run. This variance often allows the rejection of good new individuals early in the run while later accepting *only* the very best. In contrast, the Genetic Algorithm (GA) uses models of sexual reproduction; a number of parents (two or more) creates a number of children (one or more). The individuals (a population size greater than one) are evaluated for their fitness toward solving a problem. Some individuals (also called chromosomes) are selected for reproduction to create the next generation of the population.

Most genetic algorithms use the same basic step sequence:

1. Randomly initialize the population.
2. Score the population (or newest individuals of the population).
3. Check for termination criteria.
4. Select individuals for crossover.
5. Crossover those selected (reproduce).
6. Mutate those created (if desired).
7. Select individuals for replacement.
8. Repeat from step 2.

Figure 1: The Basic GA Sequence

There are many variations of nearly every step in the process. For a particular run of a genetic algorithm, a selection from these variants must be made. Collectively, these variations are called the genetic algorithm's parameters. A poor choice of parameters can slow the GA's convergence rate and diminish the final quality of its solution.

Before these steps can be taken, several important questions must be addressed:

1. How is the fitness of an individual of the population scored?
2. What are the termination criteria for the algorithm?
3. What is the solution representation for that problem?

2.1.1 Fitness Evaluation Functions

The function that evaluates a given chromosome for its fitness at solving the problem at hand is one of the most difficult to write. There are very few guidelines for writing good fitness functions (and fewer still on how to avoid writing bad ones). Improperly written fitness functions can easily mask the use of otherwise acceptable GA parameters. GAs use the score returned by the fitness functions to determine the relative merits of the chromosomes with respect to the rest of the population. Individuals with better scores tend to be rewarded by remaining in the population longer and by being given the opportunity to produce offspring. "Better" depends upon what the problem is.

Analytic functions are the most straightforward fitness functions. Once the chromosome is decoded into values for each variable they can be "plugged" into the function. The fitness is then the final solution to the equation(s). The fitness function for other problems depends directly upon the chromosome representation.

2.1.2 Termination Criteria

The termination (or exit) criterion for a genetic algorithm varies with the intent of the algorithm. Some algorithms may not know the exact termination requirements or are content to let the program "run as long as necessary". In these cases, no termination

check may be performed, and the program is forcefully terminated via the operating system when desired. In the same context, a maximal boundary may be established to limit the duration of the algorithm run. Alternatively, analysis of the genetic algorithm's runtime statistics may yield useful information for determining when it is "done." At this point, the algorithm can be terminated gracefully. One examples of this is when the average rate of progress (movement towards the global optima) falls below a given threshold (i.e. it has found a local optima). In this case, the GA is unlikely to do any better than it has, as it is not making any significant improvement. This could also be measured by a decrease in the variance of the population members.

2.1.3 Problem Representation

When using a genetic algorithm, it is necessary to decide what information is to be encoded within the chromosome. The encoded information is problem dependent. If the problem involves optimizing an analytic function, the chromosome may be one or more coordinates or variable values. For example, a function of two variables might encode the X and Y coordinates each as 16 bit fixed point decimal values. A bit chromosome (a chromosome of an arbitrarily long string of bits) of length 32 might encode the first 16 bits as the X coordinate and the last 16 bits as the Y coordinate. If the problem is an ordering problem, such as a bin packing or a traveling salesman problem, the chromosome may be a sequence of integers indicating which bin or city is to be used next. The population might consist of arrays of integers, each integer having a value [0..L). 'L' denotes the size of the chromosome.

Some problems might not utilize such uniform chromosome encodings. With a bit-based chromosome, multiple fields of irregular length may be encoded and excised using existing bit based algorithms. Only the fitness function need understand how to extract the encoded data from the chromosome.

2.1.4 Schema Theory

"The most difficult part of a random search method is to explain why and when it will work."
[MH91, p 2]

Schema Theory is used to describe and predict some parts of the behavior of genetic algorithms. Schema are sequences of information within a chromosome's encoding that represent the "building blocks" of a solution to the problem being solved by the genetic algorithm. These sequences are described using bit strings of ones, zeros and asterisks (for "don't cares"). For example the string 1 * * * * 0 is representative of all six-bit strings beginning with a one and ending in a zero. Schema are classified by length and order. The length of schema is the distance between the first non-asterisk of the pattern and the last non-asterisk of the pattern. The order of schemata is the number of non-asterisks in the total pattern. For example, the schemata * * * 1 1 * * 0 0 * * has a length of six with an order of four. Schema theory is very applicable to predicting and explaining the effects of the crossover operators. Although schema patterns are described using ones and zeros, the schema patterns are not strictly bit string patterns. These patterns apply to all chromosome types and can easily be applied as index masks onto chromosome field indices.

2.1.5 Initialization and Population Sizing

The initialization step involves iterating through each member of the population and randomly filling in their genetic material. How this is done depends upon the chromosome itself. If the chromosome is a bit representation, each bit must be individually set (or cleared). If instead the chromosome is a primitive data type (e.g. an integer or floating point data type) each gene in the chromosome can typically be set directly from the local random number generator.

The population size (N) may well be the most important parameter to a GA. Determining it has been the subject of much research and debate. Many research efforts have investigated the "proper" determination of population size. The fact is population

size is integrally linked to the selection and crossover methods and is difficult to specify correctly for an arbitrary problem. In general, smaller population sizes tend to become homogenous more quickly than larger populations. This is believed to be a side-effect of the crossover method not having enough genetic material to work with because of the inability of the population to adequately sample the problem domain. However, the effectiveness of the crossover method is reduced with the use of "large" populations (the crossover method discussion is in section 2.1.7). Suffice to say the population must have sufficient (statistically significant) coverage of the building blocks of a good solution. Without this, a GA has little probability of arriving at a quality solution.

2.1.6 Selection Methods

2.1.6.1 Fitness Proportional Selection

There is a variety of selection methods described in the genetic algorithm literature [MM96, pp. 166-171]. The most commonly used is the Fitness Proportional Selection (also called Roulette Wheel Selection or RWS) method originated by Holland. Fitness Proportional Selection conceptually assigns each individual in the population an arc of a circle proportional to the individual's fitness relative to the sum of all finesses. A random value less than the sum of all scores in the population is then generated for each selection attempt. The individual scores are summed until the sum exceeds the generated value. The individual that causes the sum to exceed the generated value is selected for use as crossover candidate. Individuals with better fitness will obtain larger portions of the circle increasing their probability of being selected. This algorithm is summarized as code in Figure 2.

```
dTotalSum = Σ_{i=0}^{N-1} adScores[i] ;

for ( iSelect = 0 ; iSelect < nNumToSelect ; iSelect++ )
{
        dRunningSum = 0.0 ;
        dThreshold = Rand() ;/* spin the wheel */
        dThreshold *= dTotalSum ;   /* scale the spin */

        for ( iIndex = 0 ; dRunningSum < dThreshold; iIndex++ )
        {
                dRunningSum += adScores[ iIndex ] ;
        }

        Select( iIndex ) ;
}
```

Figure 2: Basic Roulette Wheel Selection Algorithm

There are two common modifications to the RWS algorithm that reduces its selection pressure: rank scaling and sigma scaling. Both have the effect of reducing the merit of any single individual with respect to the population as a whole thus evening out the probability of selection for each chromosome (reducing the overall selection pressure) and decreasing the convergence rate of the GA.

RWS with rank scaling modifies the fitness scores used by the basic RWS algorithm by replacing each chromosome's score with a value (0 to L-1). Each chromosome's score is indicative its relative rank compared to the rest of the population. The following equation details the scaling:

$$\forall i \,|\, adNewScores[i] = dMinScore + (dMaxScore - dMinScore)\frac{Rank(i) - 1}{N - 1}$$

Figure 3: RWS with Rank Scaling Equation

RWS with sigma scaling also modifies the scores that are used by the basic RWS by replacing the each chromosome's score with the output of the following equation based on the standard deviation of the population (σ):

If σ is not equal to 0

$$\forall i | adNewScores[i] = 1 + \frac{adScores[i] - \overline{adScores}}{2\sigma}$$

Otherwise

$$\forall i | adNewScores[i] = 1$$

Figure 4: RWS with Sigma Scaling Equation

2.1.6.2 Tournament Selection

An alternative to the classic RWS method is the Tournament Selection method. This method is somewhat simpler than the RWS method. A tournament is "held" amongst a randomly selected sample of individuals of the population. The tournament selects the most fit individual of the sample for reproduction. The tournament size (τ) has clear implications upon the selection pressure of the GA. The larger the tournament, the greater the pressure on the population to converge. A tournament size of one is a totally random selection method. A tournament size of two has properties similar to RWS with rank scaling. For this size and all tournament sizes greater, an individual's score is irrelevant to its selection. Instead, only its rank within the population as a whole is considered. Lesser rank will always "lose" to higher rank. Duplicate scores are resolved arbitrarily. Larger tournament sizes are more elitist in nature. This increases the probability of the best individual in the population being selected and thereby increasing the selection pressure of the GA.

2.1.7 Crossover Methods

Crossover is the primary mechanism by which genetic algorithms create new (and hopefully better) solutions. This is done by scrambling the genetic material of one or

more parents to create one or more children. There are many different crossover methods available. Some are general purpose, others problem specific. The "normal" biological rules for sexual reproduction need not apply to these methods. One or more parents can generate one or more children. It is strictly up to the user or programmer. Often, at least two parent chromosomes are selected to create one or two child chromosomes. This thesis only utilizes and discusses crossover methods that involve two parents yielding two children.

2.1.7.1 Bit Crossover Methods

Perhaps the most common bit crossover method is single point crossover. A random point (p) is selected along the length of two parent chromosomes. The first of two children is created from the parents by copying *parent1[1..p]* to *child1[1..p]* and *parent2[p+1..N]* to *child1[p+1..N]*. Similarly, the second child is created by copying *parent2[1..p]* to *child2[1..p]* and *parent1[p+1..N]* to *child2[p+1..N]*. Schema analysis of single point crossover indicates that this crossover method tends to preserve longer schemata (compared to double point and uniform crossover).

Single point crossover at index 3	
Parent 1 is	AAA | BBBBBB
Parent 2 is	CCC | DDDDDD
Child 1 becomes	AAA | DDDDDD
Child 2 becomes	CCC | BBBBBB

Figure 5: Single Point Crossover Example

A similar method is double point crossover. Instead of a single crossover point, two crossover points are selected. Two children are created by copying each parent directly into a child chromosome and swapping the inner substring delineated by the two crossover points. This crossover method tends disrupt larger schemata but preserve schemata with actual values (non-*) near their ends (e.g. 1 0 0 * * * * 1 0).

Double point crossover at indexes 3 and 7	
Parent 1 is	AAA|BBBB|CC
Parent 2 is	DDD|EEEE|FF
Child 1 becomes	AAA|EEEE|CC
Child 2 becomes	DDD|BBBB|FF

Figure 6: Double Point Crossover Example

It is not hard to rationalize a K-point crossover method (where $0 < K < N$). As $K \rightarrow N$, the crossover method will preserve progressively smaller schema.

The uniform crossover method preserves smaller schema in a manner similar to a high-order K-point crossover method (where $K \rightarrow N$). Two children are created by iterating through the length of the chromosome and copying bits into the children from the parent. For each bit position, a random number is generated and thresholded. If the value is over the threshold, the bits are copied directly into the children from the associated parent. If the value is less than the threshold, the bits from the parents are swapped before being written into the children.

Parent 1 is	**A**A**A**AAA**A**A
Parent 2 is	**B**BB**B**BBB**B**BB
Child 1 becomes	**B**A**B**AAA**B**AA
Child 2 becomes	**A**BA**A**BBB**A**BB

Figure 7: Uniform Crossover Example

Research into the usefulness of uniform crossover has shown that there are two specific situations where it will tend to out-perform both the single or double point crossover methods:

1. During the end of the GA's run, when the population tends to be more homogeneous, uniform crossover can provide enough disruption during the crossover operation to

push the population further. This can be likened to the effect of mutation (section 2.1.8).

2. When a run uses a population size that is too small for adequate coverage of the searchable space. The disruption caused by the uniform crossover operator can "help overcome the limited information capacity of smaller populations and the tendency for more homogeneity." [DJS90, p. 8]

2.1.7.2 Permutation-Based Crossover Methods

For chromosomes that encode permutations, it is not possible to randomly change values in the chromosome. To do so would disrupt the uniqueness of the chromosome's genes. The following crossover methods each provide a different way of recombining the genes of two parent chromosomes to generate two new and different (usually) children.

Cycle Crossover creates child chromosomes by first copying each parent into one of the children and then randomly interchanging one of the genes. Since the chromosomes are unique, this interchange creates a redundant gene in each of the child chromosomes. The cycle crossover method gets its name from how it resolves this duplication; it chooses one of the child chromosomes and searches for the newly duplicated gene. When found, that index is also interchanged with the other child chromosome. This second interchange may remove the duplication or it may introduce a different duplicate gene. Cycle crossover repeats this process on the selected child chromosome until either there is no more duplication or the original parent chromosomes are recreated.

Schema analysis shows that cycle crossover can preserve and destroy both short and long schemata depending upon what cycles exist between the two parent chromosomes.

Parent 1 is	A	**C**	G	H	D	B	F
Parent 2 is	B	**G**	A	D	F	C	E
Pick index 2 and correct child 1:							
Child 1 becomes	A	**G**	**G**	H	D	B	F
Child 2 becomes	B	**C**	**A**	D	F	C	E
Child 1 becomes	**A**	G	**A**	H	D	B	F
Child 2 becomes	**B**	C	**G**	D	F	C	E
Child 1 becomes	**B**	G	A	H	D	**B**	F
Child 2 becomes	**A**	C	G	D	F	**C**	E
Child 1 becomes	B	G	A	H	D	**C**	F
Child 2 becomes	A	C	G	D	F	**B**	E

Figure 8: Cycle Crossover Example

Ordered Crossover works differently from cycle crossover. The ordered crossover method selects two random indexes to create a substring from each parent chromosome. Each substring is copied into the beginning of the associated child chromosome. The middle of each child chromosome is constructed by appending any non-duplicate gene from the other child chromosome. The remainder of each child chromosome is generated by iterating through the parent chromosome starting to the right of the substring (using modulo arithmetic if necessary) and appending any genes that do not already exist in the child chromosome.

Parent 1 is A B C | D E F | G H
Parent 2 is G H F | E D C | B A

Randomly pick indexes 3 and 6

Copy the substrings to the beginning of the associated child:

Child 1 becomes D E F | x x x x x
Child 2 becomes E D C | x x x x x

Fix the middle of the children:

Child 1 becomes D E F | C | x x x x
Child 2 becomes E D C | F | x x x x

Finish the rest:

Child 1 becomes D E F | C | G H A B
Child 2 becomes E D C | F | B A H G

Figure 9: Ordered Crossover Example

Schema analysis of ordered crossover reveals that schemata short enough to exist within the substring will be preserved while all other schema will tend to be destroyed.

Partial Matched Crossover (PMX) provides functionality that is a combination of the two previous methods. Like ordered crossover, two randomly located indexes are used to mark substrings of two parent chromosomes. The left portion of each child is created from checking for the existence of each parent's left genes in the opposite parent's substring. If a gene is found, the gene in the child's associated parent's substring at the same index is used. Otherwise, the gene is copied directly into the child. The right portion of each child is formed similarly. The middle portion of each child is formed by copying the opposite parent's substring into the child.

PMX effectively preservers entire schema as ordered crossover does but also disrupts schema as cycle crossover does. This provides the unique property of being able to

preserve useful schema while pushing the population forward via the recombination of other more marginally preserved schema (those partially destroyed by the Cycle-like activities).

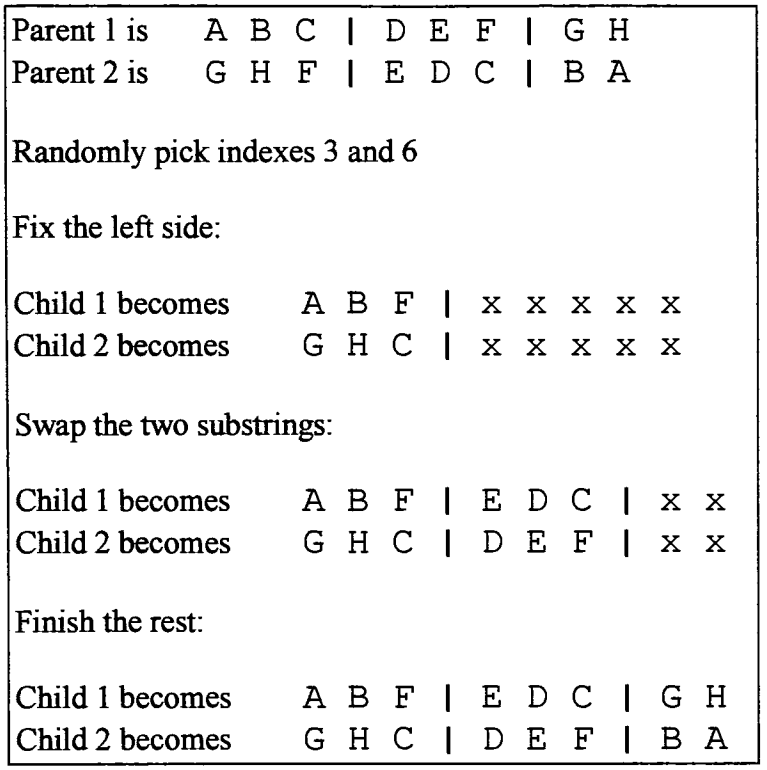

Figure 10: Partial Matched Crossover Example

Other crossover algorithms exist for specific problem instances. Maximal Preservation Crossover (MPX) has specifically been developed for (and successfully applied to) solving Traveling Salesman Problems [MH91, p. 331]. Its intent is to maintain subtours common between two parent chromosomes. (This thesis has chosen not to investigate the application and development of problem specific crossover methods. Rather, its focus is on the benefits of different parallel genetic algorithm implementations.)

2.1.8 Mutation

Mutation is the mechanism whereby the genetic algorithm extends its current genetic material by randomly changing pieces of chromosomes. This process can allow an individual of the genetic algorithm's population to move away from the local optima

towards which the population is currently climbing. Mutation is often considered an optional step in the general GA process. It is suggested that given a sufficiently large population, enough coverage of the problem domain can be made such that crossover by itself can sufficiently arrive at the best solution. While this may be true for smaller problems, problems of any "real-world" size would require populations that are impractical to work with. The GA parameter ρ_m is used to signify the percentage of genes of a specific chromosome that are selected for mutation.

2.1.8.1 Bit Chromosome Mutation

For bit-based chromosomes, the concept of mutation is straightforward. Multiply ρ_m by L to get the number of genes to mutate (M). For all M, randomly select a gene for mutation and randomly reset the bit value. The new bit value can be obtained by thresholding a value extracted from a random number generator. The threshold is used to determine the binary value of the new bit. Commonly set to 50%, the threshold can be placed anywhere in an attempt to bias mutation towards a certain outcome.

2.1.8.2 Permutation Chromosome Mutation

Simple integer chromosomes can be mutated in a manner similar to the bit chromosome. Instead of changing a bit, a completely new integer is extracted from the local random number generator to replace the targeted gene. This method can potentially result in the duplication of values. Permutation representative chromosomes require a slight modification to the mutation concept. Since only one instance of each gene is allowed to exist in the chromosome, randomly changing a gene's value would result in the duplication of an existing gene. This is obviously undesirable. Instead, the mutation rate (ρ_m) is divided by two to create the modified mutation rate ρ_{mm}. ρ_{mm} is multiplied by L to get the number of genes to mutate (M). For all M, an index randomly selected to mutate. A new value is generated for this gene and the chromosome is searched for the index of the gene that has this newly obtained value. When located, the two genes are swapped. The normal mutation rate is halved because this mutation algorithm changes two genes not one as the bit mutation algorithm does.

2.1.9 Replacement Policies

2.1.9.1 Generational

The classic GA creates an entirely new population from the repetitive selection of members from the original population. This is considered a generational model because each new population is considered to be a generation. Variations of this policy exist. A common variant is the replacement of only some percentage of the population. The replacement percentage is indicated by ρ_r. Often, $\rho_r * N$ (where ρ_r is small) is replaced instead of the entire population, and those replaced are commonly the least fit individuals of the population. Another variant accommodates the lack of preservation by the selection methods. Elitism forcibly requires some percentage of the best individuals in the population to be retained into the next generation. While elitism normally retains only one or a few individuals, this value could also be viewed as a large replacement percentage. The classic GA replacement model could similarly be stated as 100% replacement or 0% elitism.

2.1.9.2 Steady-State

Instead of replacing the entire population at once, the steady-state GA replaces only a few individuals at a time. This helps maintain stability within the population and results in a built-in elitism. The best individual of a population will never be selected for replacement. Additionally, this mechanism does not have "generations" of individuals. Subsequently it is not possible to compare a steady-state GA directly to a generational GA. Instead, comparisons must be based upon the number of fitness evaluations or fitness comparisons executed by the algorithm. In this way, the relative merits of each GA type can be analyzed. This policy allows for a built-in niching mechanism because the selection of replacement individuals usually involves the random, but directed, selection of lesser fit individuals.

2.1.10 Premature Convergence

One of the most significant problems facing the GA community is the premature convergence of a GA to a suboptimal value. This problem has been the subject of much research and debate. Genetic diversity (variances in the population's genetic material) is

a significant component of the success of genetic algorithms. The fitness proportional GA "assigns exponentially increasing numbers of trials to the observed best parts of the search space." [SFP93, p. 128] This ability comprises much of the GA's capacity to solve problems. At the same time, this can restrict its ability to search a solution space by reducing the total amount of genetically diverse material in the population. Typically, an effort to reduce the possibility of premature convergence also slows the convergence rate of the GA in general. This is usually accomplished by reducing the selection pressure of the GA. A trade-off, then, exists between the need to find a good solution quickly, and the slowing of the convergence rate of the GA (which improves the potential of creating an even better individual that might not otherwise been realized).

Algorithms to reduce the seriousness of this problem abound. DeJong introduced the concept of Population Crowding in 1975 [SFP93, p. 129]. After creating a new individual for the population, an individual is selected for replacement by first randomly drawing of some percentage of the population and then determining which of the members of the drawing most closely resembles the new individual. This scheme maintains genetic diversity by enforcing the "uniqueness" of each member of the population. Crowding is typically implemented in a steady-state GA (see section 2.1.9.2). Fitness Sharing works similarly. However, it penalizes new individuals for the existence of population members similar to them. The usefulness of fitness sharing can be limited by the loss of genetic material within the population as an effect of the selection and crossover methods. This can be compensated for by increasing the population size. Unfortunately, fitness sharing is an order N^2 algorithm requiring rapidly increasing amounts of time. The determination of the penalties for fitness sharing also requires knowledge of the problem domain that may not be available. Fitness sharing can still be useful, however, because it does force the existence of stable niches rather than only slowing the convergence rate of the algorithm.

2.2 Parallel Genetic Algorithms

There are three major categories of parallel genetic algorithms: global, coarse-grained, and fine-grained. This section presents an overview of each type along with several alternative designs originating as coarse-grained GAs.

2.2.1 Global Parallel

Global Parallel (or Panmictic) Genetic Algorithms are implementations that strictly are parallelizations of the serial GA. Assuming a fitness proportionate selection method (see section 2.1.6.1), the fitness evaluations and mutation of individuals of the population's next generation can all be done in parallel. A bottleneck in the parallel algorithm occurs only when it is necessary to calculate the population's average fitness value and to sum the entire population's fitness' With some planning, selection, crossover and replacement can also be done in parallel. The benefits of these implementations, however, are not necessarily worthwhile. The genetic operators (e.g. crossover, selection, mutation) are quite simple, and it should be expected that the communication overhead required to parallelize these operators could easily negate or penalize the speedup or other performance gains normally obtained through the parallelization effort. Nevertheless, if the evaluation of the fitness function is computationally intense, globally parallel GAs are simple to implement and can be more efficient than other parallel methods.

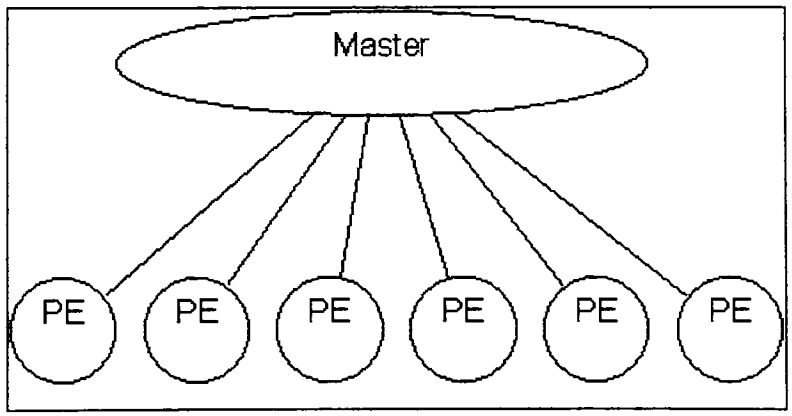

Figure 11: Example Global Parallel GA

2.2.2 Coarse Grained Parallel

Coarse Grained Parallel Genetic Algorithms (CGGAs) are often referred to as Distributed or Island Model GAs. This comes from the fact that CGGAs are multiple serial genetic algorithms running independently. The separate serial GAs are called demes. All mutation, crossover, and selection operations are performed as previously detailed in section 2.1. Unlike globally parallel GAs, however, the genetic operators are applied to their local deme and not to the population as a whole.

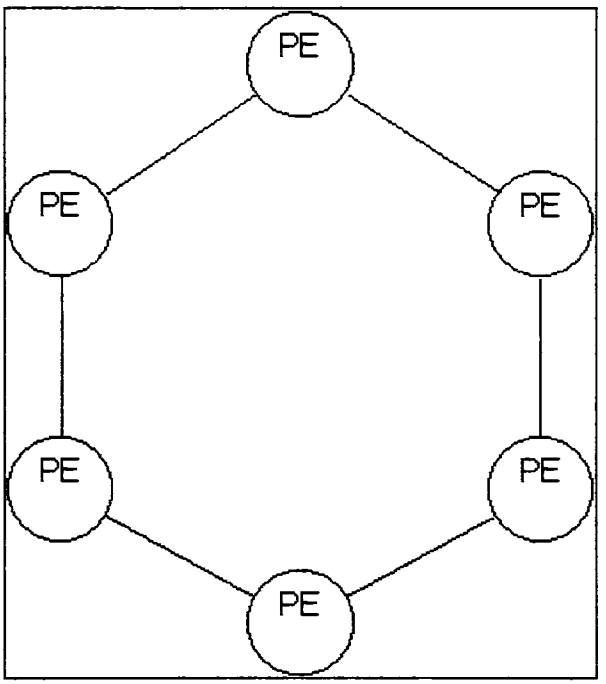

Figure 12: Example Coarse-Grained Parallel GA

A connection topology for communication must be established for the various demes of the CGGA. Often static (although not required to be so), the connection topology determines to what demes any single deme may migrate individuals to (including itself). The intent is to allow each deme to move through the problem domain somewhat independently but with periodic updates (migration) from the global population at large.

Unfortunately, any amount of migration will eventually cause the global population to converge.

CGGAs have three additional parameters to specify: the migration rate, the migration interval and whether or not the demes are synchronized at the time of the migration interval (deme synchronization). The migration rate is the percentage of the deme's population that is pushed to each of its peer demes. A value of 0% effectively isolates all demes from each other. That is, each deme develops totally independently from each other. It is important to note that the percentage of the population potentially replaced due to immigration is not the same as the emigration percentage. In general, the emigration rate is the sum of the number of immigrants from each connection to the deme (this expression allows for asymmetric topologies with uneven migration rates). The connectivity of the demes has also been the subject of debate. Some studies have shown that a high degree of connectivity between demes is best while others have demonstrated contrary results. It is clear that a few peers each with a large migration rate can easily displace the entire local population in a single step. Therefore, the migration rate and network topology must be carefully considered. The migration interval determines when each deme broadcasts its emigrants to its peers. Longer intervals allow each population to develop their own localized niche of the problem domain.

The deme synchronization parameter determines if the demes stop and wait for a "CONTINUE" signal after broadcasting their emigrants. There are several important characteristics to consider when synchronizing demes.

- All demes develop their populations at the same rate allowing for an accurate tracking and recording of deme statistics. However, if a heterogeneous computing network is utilized, the benefits of the fastest machine will be eliminated as that machine will be forced to wait for the slowest machine to complete its task.
- Depending on the implementation, runs of synchronized demes can often be a repeatable. Variations in network communication efficiency become the only impediment to complete repeatability.

- It is possible for the deme implementation to know that all immigrants have been received once the "CONTINUE" signal has been received. Some useful optimizations may be introduced utilizing this fact.

When the demes are not synchronized:

- Heterogeneous networks are allowed to work at their own pace: fast machines are not limited by slower machines maximizing the effective of the faster machines.
- The demes are free to evolve freely. It is not dependent upon the reception of peer and therefore can continue as if it was simply a serial GA with input from other peers at an unquantifiable and potentially varying rate.
- If a deme has more than one peer, the recreation of the run may be impossible, the communications delays may not be repeatable.

2.2.3 Fine Grained Parallel

Fine Grained Genetic Algorithms (FGGAs) are similar to the CGGA in that demes are used and the genetic operators are applied only at the deme level. The demes for this architecture are small; as small as possible. The ideal situation for a FGGA implementations is to have a single processing element for each individual in the global population. In this case, the concept of a deme is extended to become a set of processors. These parallel GAs are typically implemented on massively parallel machines. It has been shown that the size and shape of the deme alters the overall selection pressure of the GA. The selection pressure can be directly affected by the ratio of the radius of the deme to the radius of the hardware's connection topology [SDJ96, p. 243].

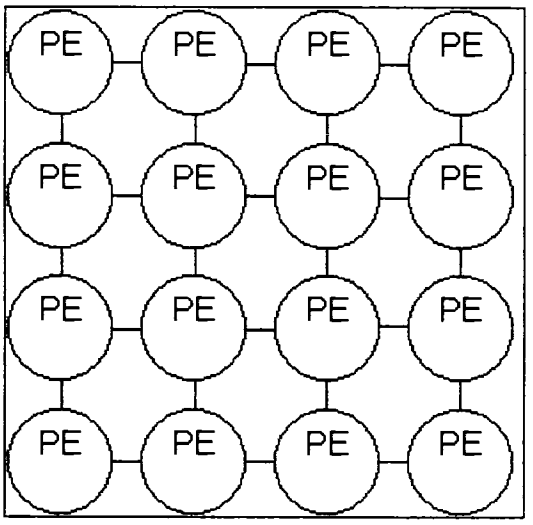

Figure 13: Example Fine-Grained Parallel GA

2.2.4 Hybrids

Some research has been focused upon the combination of these three classifications of parallel GAs. Usually, a small global GA or a FGGA is contained within the deme of a larger CGGA (less commonly, a CGGA replaces the deme's serial GA). The deme gains the benefits of the GA model that replaces the serial GA and correspondingly, the CGGA as a whole gains. [CP971] shows that there are limits to the number of processing elements that can be effectively utilized by a globally parallel GA or by a CGGA. These hybrids are a wady to utilize more processing elements within the limitations of the respective designs and potentially improve the both the results obtained and convergence rate of the GA. Indeed, hybrids may be the best mechanism for solving very large problems. The CGGA deme does not have to be a GA at all. It could be a Finite Element Analysis (FEA) algorithm that iterates for a period of time simulating the migration interval or a completely separate node that only feeds the CGGA new best solutions as it finds them. A combination of FEA and a CGGA has been successfully used by Michigan State University's (MSU) Genetic Algorithms Research and Applications Group (GARAGe) to optimize the mass of a composite-based flywheel [ED97, p. 1]. The results of this hybrid approach (along with other GA related inventions) has significantly outperformed more basic CGGAs.

2.2.5 Non-Uniform Distributed

The Non-Uniform Distributed Genetic Algorithm (NUDGA) attempts to improve upon the basic CGGA. For most problems, it is difficult to know what parameters the genetic algorithm needs for maximal efficiency. NUDGA initializes each deme to different randomly generated GA and CGGA parameters. The intent is that despite the lack of knowledge of the "correct" parameters, some combination of mixed parameters will yield better results than the CGGA with every deme initialized to the same parameters.

2.2.6 Adaptive Distributed

> *"An EA, whether serial or parallel can be effective only when a proper balance between exploration (via well chosen parameters) and exploitation (well chosen selection pressure) is maintained."*
> [SDJ96, p. 236]

It is clear that the selection/optimization of the GA parameters for an arbitrary non-trivial problem is difficult particularly because of the mutual interactions of the parameters themselves. This problem is even more apparent if the problem domain is not well understood. The CGGA is simply an extension of the basic GA; adding even more parameters to the already large and varying set that are the basic parameters. While the NUDGA has the potential to achieve better results than the CGGA it can (and does) choose sub-optimal parameter sets. Having no means of identifying or correcting those poor choices, the user becomes responsible for monitoring the progress of the NUDGA and making the decision to prematurely terminate the experiment. The Adaptive Distributed Genetic Algorithm (GA2) extends the NUDGA by this one more step; the GA2 uses a supervisory serial GA to measure the progress of each deme and to alter the parameters of specific demes which it decides to. A population size equal to the number of demes is utilized where the chromosomes of the supervisory GA encode each deme's GA and CGGA parameters. All GA operators are exercised on the population. When replacement is performed, the demes that correspond to the chromosomes being replaced are sent the GA and CGGA parameters encoded in the new replacement chromosomes. Herein lies the adaptive nature of the GA2. When one deme significantly outperforms the others, that deme becomes a stronger candidate for selection and a lesser candidate

for replacement (per the policies of the selection operator and the GA style). No fitness function has been clearly identified as optimal for the GA2. However, some are obviously better than others. [WGP97] has identified the following function as the most promising to date:

$$\frac{\textit{Deme's best score}}{\textit{\# of fitness evals over G generations}}$$

Figure 14: A good GA2 fitness function

Figure 14 shows that unless a deme continually improves its best score, eventually the deme's score will "flatten" and become a candidate for replacement. It is interesting to note that the chromosome's score does not come directly from the chromosome itself, but indirectly from the decoding of the chromosome and execution over some period of time.

GA2 relies upon the periodic synchronization of the demes. When the last deme has reported its current progress, the supervisory GA suspends main CGGA algorithm before it can release demes from their synchronization. The suspension is until the supervisory GA has decided which demes need new GA parameters and has pushed them to the selected demes. The selected demes immediately adopt the revised parameters. The supervisory GA then suspends itself after releasing main CGGA algorithm.

Much of this design and analysis comes from the research done by the MSU GARAGe and their Distributed Adaptive Level-two Genetic Algorithm (DAGA2). DAGA2 is an instance of the nGA described in [WGP96]. nGAs extend GA2 by indefinitely adding layers of GAs. This allows for the evolution of GA structures. Such evolved structures may have useful properties that might not normally be realizable.

2.2.7 Island-Injection

Another variation of the CGGA is the Island-Injection Genetic Algorithm (iiGA). The iiGA has several interesting features. First, the topology of the network is commonly a

tree or rectangular mesh and the demes are not necessarily synchronized. This allows maximal application of the second feature: the chromosome encoding differs from deme to deme. Deme nodes higher in the network tend to have coarser, less defined or compressed chromosomes. This allows for either faster evaluation of the chromosome as portions either can be ignored or are known, or directs the deme to evaluate only a portion of the solution space as dictated by the untouchable areas of the chromosome. The best solutions are passed down the network to the next "layer" of demes. Proceeding down the network, increasingly more of the chromosome becomes available for manipulation by the associated GA. This technique is new but has been successfully applied to a number of problems.

3. Implementation Overview

3.1 Java

The Java programming language originally was developed by Sun Microsystems in 1990 for embedded systems programming. It was intended to be an architecture-neutral language that could be run on most microprocessors existing at the time, and any future ones. While it has not yet been fully adopted as an embedded systems language, it has inserted itself as a viable middle-ware alternative to C and C++. Java has two significant characteristics that make it useful to both the embedded systems programmer and to the applications programmer: simplicity and portability.

3.1.1 Simplicity

One of the goals of Java designers was to create a simple language. To assist developers in learning the new language, the syntax of Java was based on that of C and C++. All of the control structures remained the same with the exception of the *goto* keyword. It was removed. The C++-like *try* - *catch* clauses necessary for exception handling also have been included and like Ada, exception handling has become an important part of the language. In this manner Java significantly exceeds its C++ parent. It is difficult to write a Java program without working with exception handling of some kind. Java requires the specification of exceptions thrown within a method as part of the method signature. Failure to do so results in a compiler error. C++ does not yet require exceptions to be specified in the method signature, although it is a proposed change to the ANSI standard. Java also has no preprocessor support. Instead, an "import" statement is built into the language to facilitate inclusion of other packages and classes. Not including a C-like preprocessor indirectly eliminates the use of the #define C preprocessor statement which is error prone and commonly misused by developers. Pointers too have been eliminated. This action removes a significant source of coding defects pervasive throughout the software industry. Java retains the common C language primitive data types (e.g. *char*, *short*, *int*, *long*, *float*, and *double*) but includes the useful *boolean* and *byte* types as well. All integer types are signed with the exception of *char*. When used as parameters to methods, primitive data types are sent using the "pass-by-value" protocol.

Java is an object-oriented language. It supports the definition and instantiation of classes similar to C++ and Smalltalk. The most noticeable difference Java has in this respect is the lack of support for multiple inheritance. The designers of Java considered multiple inheritance unnecessary and cumbersome. However, since the ability of a class to exhibit multiple class characteristics *is* desired, the language designers developed class *interfaces*. Java interfaces are similar to C++ pure abstract classes. That is, a Java interface only specifies the methods that an implementation class must export. No code can be contained with an interface. An arbitrary Java class can implement multiple interfaces, but only can be derived from a single class. This simplifies both the concepts of object design and the implementation of the language. Unlike primitive data types, when objects are used as a parameter to a method, the "pass-by-reference" protocol is used.

Java also contains threading as part of the standard support libraries. There are two ways to utilize the threads. First, it is possible to directly subclass the java.lang.Thread class and then override the *run* method. However, this couples code to be executed in the thread directly with a specific thread implementation. The second mechanism is to create an object that implements the java.lang.Runnable interface. One java.lang.Thread constructor accepts a Runnable as a parameter. This decouples the code to be executed from any specific subclass of java.lang.Thread. To facilitate thread safety, Java also supports method and object synchronization with the use of built-in monitors. By qualifying a method as *synchronized*, only one thread of execution is allowed in the method. Using the synchronized keyword upon an object, Java also provides the ability to protect critical sections of code. If the monitor is unavailable for the synchronized object, that thread of execution is suspended until the monitor does become available.

3.1.2 Portable

Java source code is "compiled" into a pseudo-code called byte-codes. Byte-codes form the basis for Java's portability, and are instructions for an abstract machine called the

Java Virtual Machine (JVM). The JVM is an interpreter that executes the byte-codes on the target hardware. The use of a JVM shields a Java developer from the operating system on which the code is executed allowing for Sun Microsystem's trademark "Write Once, Run Anywhere."™ For a significant portion of the Java language and libraries this saying is true; Java may be the most portable modern programming language to date. Unfortunately, issues still exist that divide JVM implementations and must be accommodated by developers. While interpreted languages are notoriously slow, there are some tangible benefits of this architecture: late-binding and dynamic code loading, and language security.

Late binding is a language mechanism where external code references are not committed until required at execution time. A statically compiled and linked language (such as C) requires all external references to be known and stable before the code starts execution (i.e. all references must be formally bound at the time the executable image is produced). This is not so with Java; a method call to a class is not resolved until the method is invoked. This allows for a number of interesting features - particularly dynamic code loading. Dynamic code loading is an extension of late binding where Java class files can be retrieved from remote locations.

Language security is related to late binding and dynamic loading. Since code can be indirectly loaded from remote and potentially unreliable sources, the Java language designers developed a security matrix dictating what classes, loaded under certain conditions, can and cannot do. Classes loaded locally can have complete access to the machine including reading and writing to files. Classes loaded from remote locations cannot do either (unless the local security policy has been altered). The policies are enforced by a byte-code verifier. As code is read, the byte-code verifier examines each byte-code for validity. If a code sequence violates the policy, the class is rejected. No other language has this type of built-in integrity check. This notion and the late binding concepts are fundamental to the development of Java applets and Java enabled World-Wide Web browsers.

3.2 Remote Method Invocation

Java supports the defacto-standard distributed communication mechanism (Internet Protocol sockets) within its standard library java.net. Using sockets within Java is relatively simple yet burdensome. The developer is required to recreate the communication mechanism for each project. Remote Procedure Calls (RPCs) are another means the developer has to interface with distributed systems. RPCs simulate a procedure call giving the developer a simpler abstraction to work with than the sockets (or other communication mechanism). RPCs handle the details of marshaling and unmarsheling data that is transmitted across the interface between two computers. Unfortunately, RPCs do not translate well into a distributed, object-oriented environment. RPCs do not easily handle multiple objects communicating in different process and address spaces such as those common to object-oriented distributed systems. Remote Method Invocation (RMI) is needed in these situations. RMI is implemented as a Java native mechanism with the intent to facilitate distributed computing. RMI utilizes the Proxy design pattern (see section 3.3.2.2) and relies heavily upon object serialization (which is now available with JVM version 1.1 or higher) to move information from one location to another.

3.2.1.1 The Mechanics

To utilize an object via RMI requires the definition of interfaces that extend the java.rmi.Remote interface. These interfaces define how remote references interact with the remote objects. Methods defined in the interfaces must declare that they throw at least a java.rmi.RemoteException. A RemoteException is thrown when RMI has trouble communicating with the remote instance or machine. Communication problems are common with network based applications. RemoteException is a base class (derived from java.io.IOException) for exceptions that can occur while doing network based communications.

The actual remote object is an implementation class that is derived from java.rmi.server.UnicastRemoteObject and implements one or more of the previously defined remote interfaces. An application acting as an RMI server must replace the

default Java security manager with an instance of java.rmi.RMISecurityManager. The RMISecurityManager augments the standard security manager for the specific purpose of supporting RMI.

RMI does not currently support object activation (activation does exist in the current JDK 1.2 betas). Therefore, live objects to be exported and utilized via RMI must be registered with a name service. There are two mechanism for establishing a name service. The first involves a separate deamon process called rmiserver. This deamon process must be started before the server side application is initiated. The naming service (called a registry) supports a "well-known" interface from which external objects can query to obtain information about the exported objects. The alternative mechanism is to directly create a registry by invoking the *createRegistry* method from the java.rmi.registry.LocateRegistry class. Once a registry has been established, the server application attaches a name to an object with the *bind* or *rebind* methods of java.rmi.Naming. The name attached to the object is of the form "//hostname/some_object_name" (*some_object_name* is simply an example, the developer may choose whatever name he or she wishes to define the implementation or interface class). Clients obtain references to instances of remote objects via the *lookup* method of java.rmi.Naming passing *some_object_name* as a parameter. This eliminates any ties to the actual implementation on the remote machine and is important because the object returned to the client is actually a stub class that implements a Proxy communication pattern (see section 3.3.2.2). After the remote objects are successfully compiled, and before they are fielded, another "compiler" must be executed on the remote classes. rmic generates stub and skeleton classes based upon the remote interfaces exported by a particular class. The stub classes are downloaded from the remote host and utilized by the client side to simulate instances of the remote object. The skeleton classes are used by the server side to coordinate communications. Method invocations made upon the stub class suspend the execution of the calling thread (i.e. are "blocking") until the return value is received from the remote object. Method invocations are made to the local stub instance. The stub communicates back to the

remote skeleton instance and subsequently to the actual remote object. Parameters and return values are transferred by serializing each required object. Serializing converts the instance data into a sequence of bits suitable for writing into an arbitrary data stream. The recipient of the steam may be a persistent data store or some communication medium. In the case of RMI, it is a stream tied to a TCP/IP socket.

3.3 Design Patterns

Design patterns are new innovations in object-oriented design. Design patterns present and discuss reusable object collections that have been shown to solve specific problems. One of the original publications of software design patterns [GEA95] presents tradeoffs and implications of the presented patterns. [GEA95] also breaks design patterns into three classifications: creational, structural, and behavioral.

3.3.1 Creational Patterns

Creational patterns deal with the object instantiation process. These patterns facilitate the move away from large-scale hard-coded behaviors to the more flexible aggregation of smaller, more fundamental behaviors. They also tend to hide the mechanisms by which objects are created as well as what concrete classes are being utilized by a system.

3.3.1.1 Factory Method

The Factory Method pattern defines an abstract interface for creating objects but delegates to the subclasses what object is actually instantiated. This removes the knowledge of the exact object implementations from the parent class allowing it to be more flexible (it can work with any subclass of the abstract interface). This pattern allows parallel class hierarchies to be implemented in a simpler fashion by providing uniform "hooks" for the subclasses to override or augment behaviors of the parent classes.

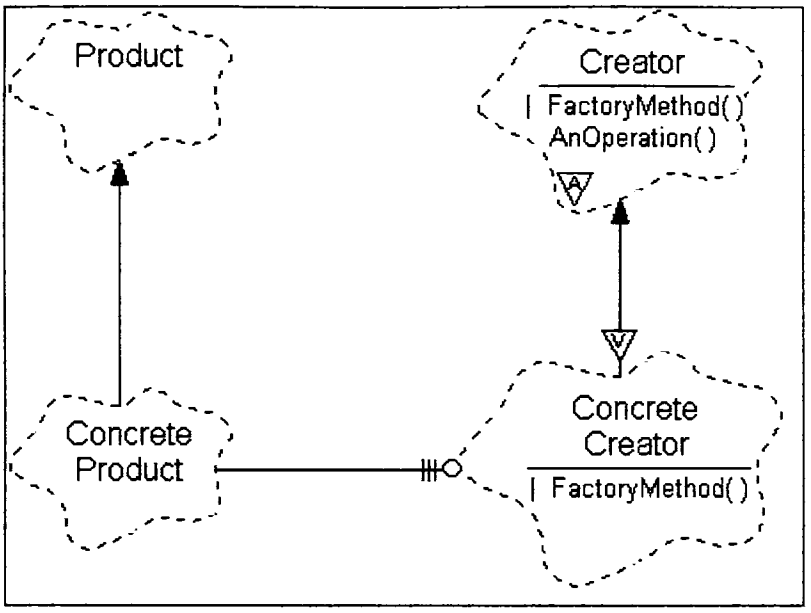

Figure 15: Example Factory Method Pattern Booch Diagram

3.3.1.2 Singleton

The intent of the Singleton pattern is to control or limit the number of instances a class allows. As the name of the pattern implies, typically the restriction is to that of a single instance. To accomplish this, the class constructor and destructor methods are hidden from the public interface. Public access clients are therefore unable to directly instantate or destroy the object. Instead, public class methods are provided as alternatives to the constructor and destructor methods.

When only a single instance of a class is allowed, a class variable can be used to maintain a reference to the single instantiation. Invocations of the replacement constructor can first check to see if the class variable is initialized. If it is not, the non-public constructor can be invoked to initialize the class variable. A reference to the class variable can then be returned to the client. In either case, the replacement constructor must maintain a reference count to the instance (the number of times the replacement constructor has been invoked). This reference count is utilized by the replacement destructor method

which decrements the reference count during each invocation. When the reference count becomes zero, the class variable is destroyed and uninitialized.

Limited resource control (e.g. only five instances are allowed) can be achieved similarly by replacing the class variable with an array or list structure. This, however, borders on an implementation that more closely resembles a Proxy pattern (section 3.3.2.2) rather than a Singleton pattern.

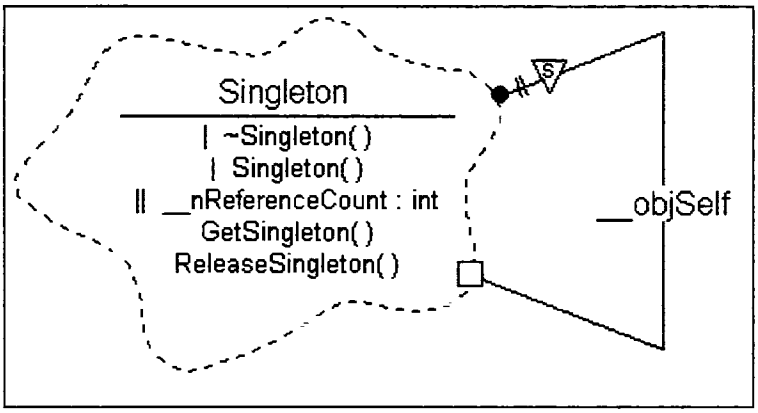

Figure 16: Example Singleton Pattern Booch Diagram

3.3.2 Structural Patterns

Structural patterns deal with the composition of object structures. Object composites are considered to have better design characteristics because they promote the development of cohesive yet loosely coupled classes.

3.3.2.1 Adapter

The intent of the Adapter pattern is to convert an inappropriate client interface into one that is acceptable. There are two forms of the Adapter pattern: class and object. The class adapter pattern implements both the desirable and undesirable interfaces. Invocations on the acceptable interface are translated into self-invocations on the undesired interface. The object adapter pattern implements the desired interface and owns an instance of class that is undesirable. Invocations on the desired interface are

delegated to invocations upon the owned instance. This pattern facilitates reuse of implementations that are imperfect for the current client.

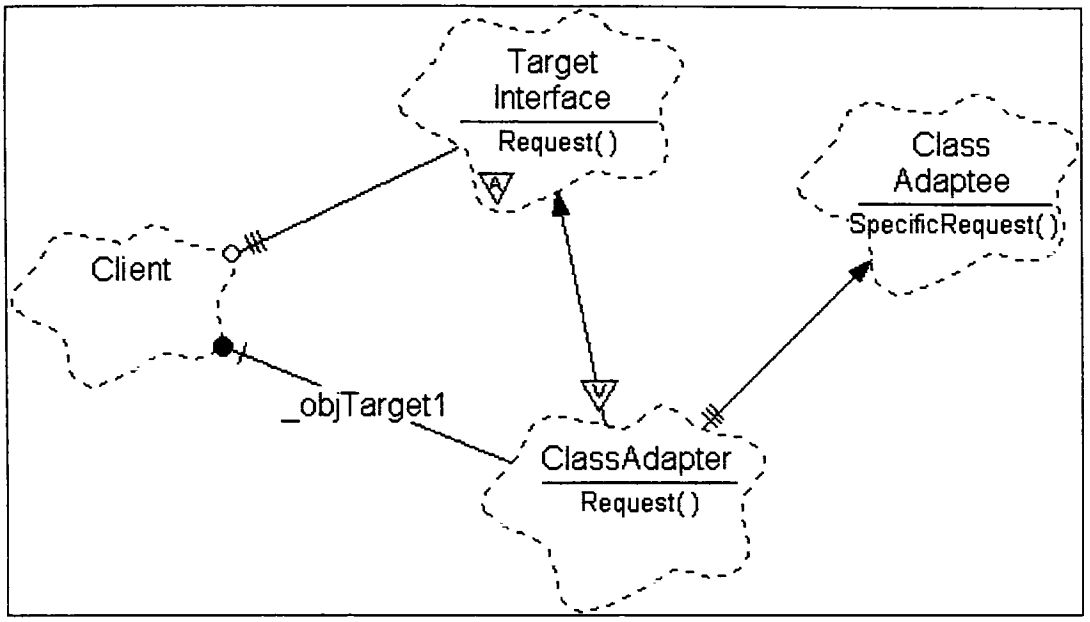

Figure 17: Example Class Adapter Pattern Booch Diagram

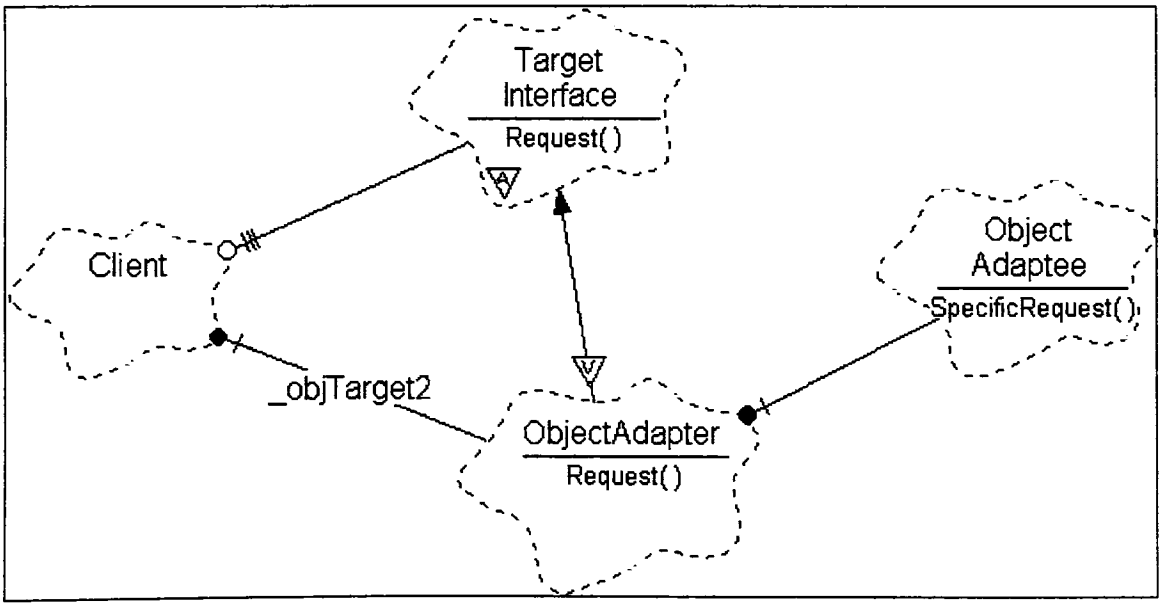

Figure 18: Example Object Adapter Pattern Booch Diagram

3.3.2.2 Proxy

The Proxy pattern provides a representative or placeholder object as a stand-in for the actual object instance. This decouples the client from the actual object or implementation providing much in the way of flexibility in both design and implementation. The proxy object implements the same interface that the actual object exports, preventing clients from determining the difference, but "owns" a reference to an "actual" object. The proxy object delegates requests from the client to the actual object. One implementation of this pattern, called a Remote Proxy, has the actual object existing in a different address space or even a different physical location. Another implementation (Smart Reference) is used as a replacement for pointers or references in general. Smart References can perform reference counting, delayed loading and even operating system level resource control when necessary.

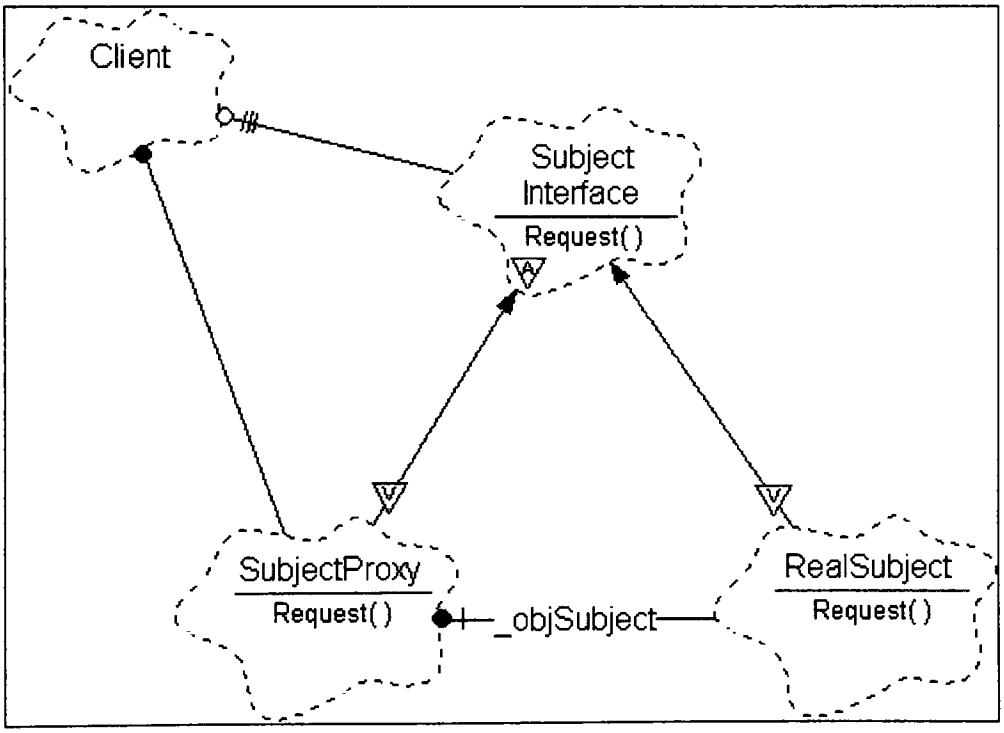

Figure 19: Example Proxy Pattern Booch Diagram

3.3.3 Behavioral Patterns

Behavioral patterns detail and describe the communication patterns of objects. These patterns help manage algorithm complexity (particularly runtime behavior).

3.3.3.1 Observer

The intent of the Observer pattern is to facilitate the decoupling of classes from a data source and from other classes. A data source supports the attaching and detaching of an abstract interface for itself. The abstract interface is implemented by clients to the data source. When the data source decides that it is necessary, it can iterate through its list of registrants invoking the abstract interface. Other than the supporting of the abstract interface, the data source is not aware of any client implementations. The clients, too, are not aware of the implementation of the data source, other than the registration interface. There are often no limitations to the number of registrants allowed to the subject. This pattern is utilized extensively by user interfaces.

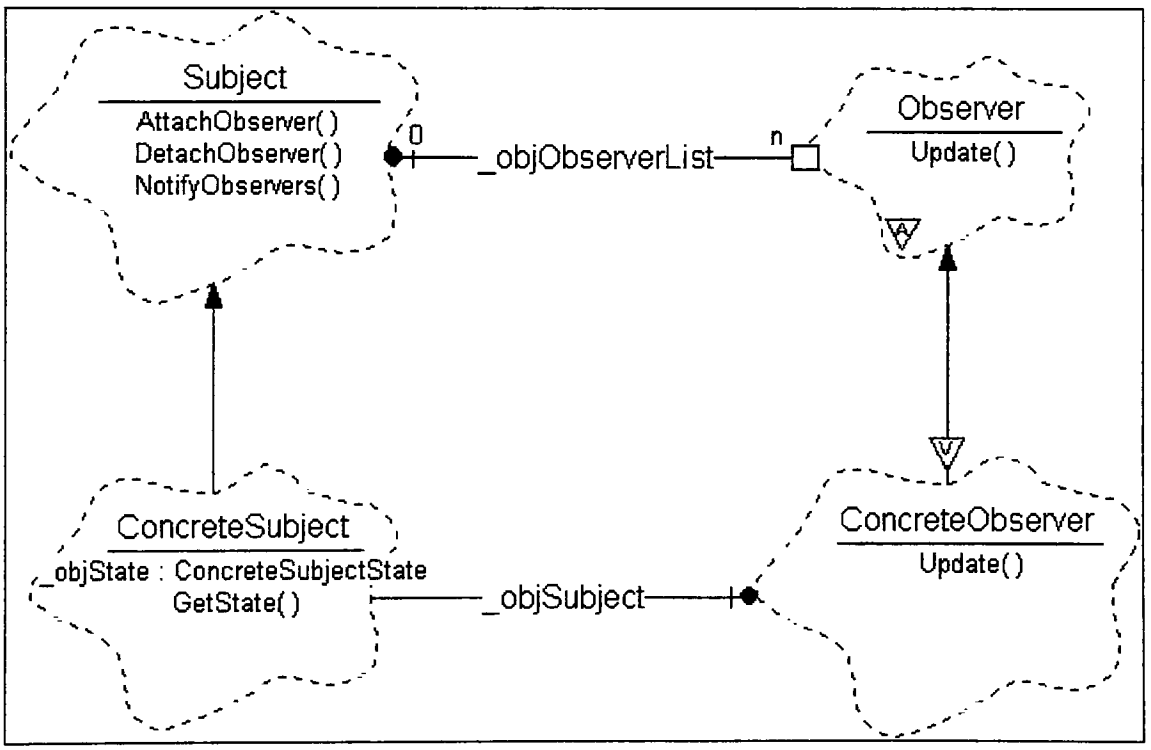

Figure 20: Example Observer Pattern Booch Diagram

3.3.3.2 Strategy

The Strategy pattern utilizes an abstract class to encapsulate an algorithm and delegate requests to that interface. Concrete classes can be derived from the abstract class which implements the interface as appropriate. This pattern allows the specification of an interface without regard to its implementation promoting loosely coupled designs and information hiding. This pattern can also reduce or eliminate the use of conditional code. Instead of having several *case* statements in the code for handling particular behaviors, the behavior specific code can be encapsulated into a subclass of the general procedure interface. Subsequently this pattern can produce a large number of small objects to be managed in a project. Further, the main implementation may neither be aware of nor understand all of the variations of implementation. This complicates the code and the utility of the implementations.

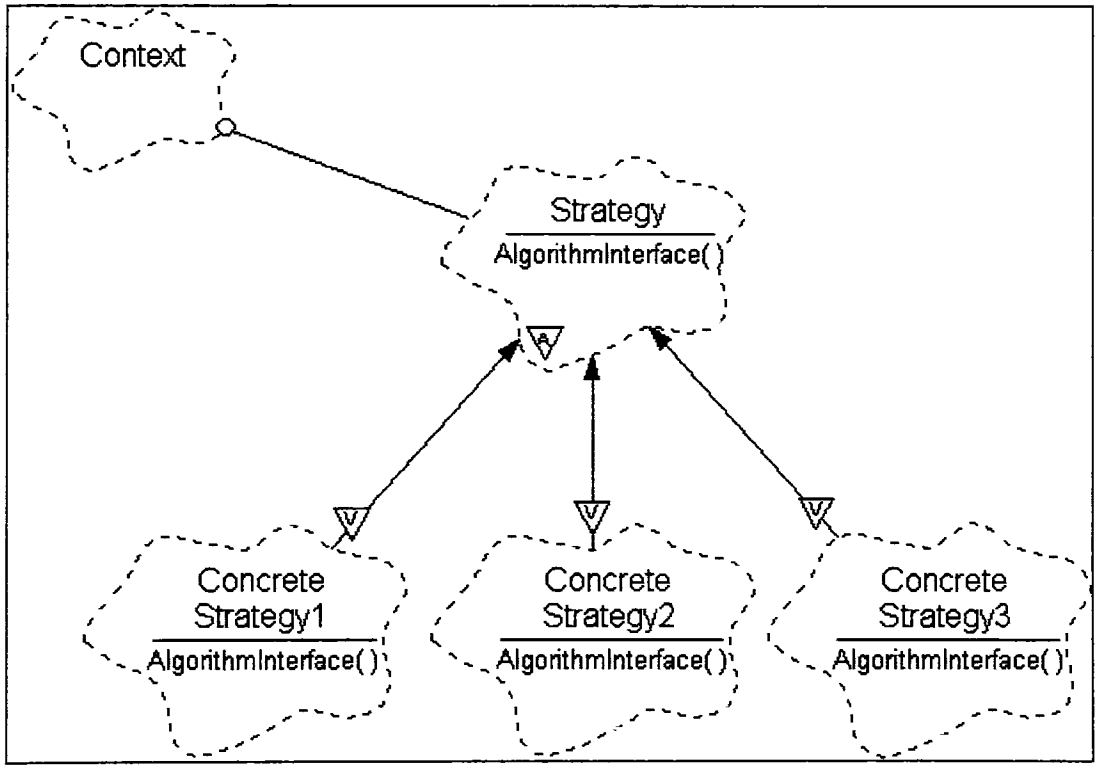

Figure 21: Example Strategy Pattern Booch Diagram

3.4 TSPLIB 95

The TSPLIB is a collection of TSPs and other similar problems compiled from many sources and maintained by Gerhard Reinelt. It has specific problems with both known solutions and known boundaries for Symmetric TSPs, Asymmetric TSPs (ATSPs), Hamiltonian Cycle Problems (HCPs), Sequential Ordering Problems (SOPs) and Capacitated Vehicle Routine Problems (CVRPs). Each problem is presented as a data file in the library. The format of a TSPLIB data file is somewhat complex but is flexible enough to represent nearly any instance of these problem types. There are two sections to the data file — the specification part and the data part.

The specification portion of the data file allows for an internal name, a description, a problem dimension, and a type of problem. A type field can identify TSP, ATSP, SOP HCP, and CVTP instances as well as a tour instance When a tour is indicated, the data portion of the file will contain one or more tour sequences instead of the normal problem data. The specification part of the data file also indicates how edge weights are calculated. TSPLIB support the specification of a number of different calculations: Euclidean 2-D, 2-D rounded up and 3-D distance, maximum 2-D and 3-D distance, Manhattan 2-D and 3-D distance, geographical distances (using coordinates in latitude and longitude), a modified 2-D Euclidean distance (denoted as ATT), and a few others. The data file can also stipulate how the edge weights and node coordinates are specified in the data portion of the file (including using a full matrix, and many variations of a triangle matrix).

The data portion of the TSPLIB data files contains lists of edge weights, edge data, demand lists, depot lists, display data, and node coordinates. A section also exists for the specification of a tour of nodes. All sections are optional, but specific sections are expected to exist for particular problem types.

3.5 Primary Packages

There were two main intentions with the architecture of this thesis: it must be easy to use to solve problems and it must be straightforward to extend for investigation into the

myriad of variations that is Genetic Algorithms. Many GA libraries contained extensive feature sets but were very difficult to use. It was the intention of this thesis that it be just as simple to implement a trivial problem such as $y = x\sin(x)$ as a more difficult problem such as the GA2's supervisory GA. Extensibility was the harder part. Software extensibility requires a significant amount of thought and planning. Extensibility requires both the ability to derive appropriate subclasses that can override essential sections of functionality as well as support the development of completely new classes gracefully. Many of these requirements can be satisfied using Factory Method patterns (section 3.3.1.1) and Strategy patterns (section 3.3.3.2).

3.5.1 Basic GA package

The edu.rit.eecc.drp.ga package provides the core algorithms and objects for using genetic algorithms. It is broken into several smaller class collections: chromosome classes, GA utility classes, selection method classes, crossover method classes and GA style classes.

3.5.1.1 Chromosome Classes

At the root of the chromosome class collection is the **ChromosomeFactoryInterface** interface and the **ChromosomeBaseClass** class. All chromosome classes must be derived from the abstract ChromosomeBaseClass as it implements the java.io.Serializable interface to facilitate the transfer of a chromosome over a data stream and the java.lang.Cloneable interface to allow a chromosome to be copied. No implementation is given to the Cloneable interface as the implementation of that interface is specific to each subclass. Two major chromosome implementations are included in the package: **BitChromosome** and **IntegerChromosome**. BitChromosome is the classic implementation of a string of bits. It is itself a complete class (i.e. instantiatable) but two subclasses (**XBitChromosome** and **XYBitChomosome**) have been derived to demonstrate the how to create application specific chromosomes. IntegerChromosome provides a basic chromosome class for using the native data-type *int* as a gene. No restrictions are placed on the contents of the IntegerChromosome. It is strictly up to the designer as to how they want to work with the class. Again, a subclass

(**UniqueIntegerChromosome**) is provided to demonstrate how an application specific subclass can be created. UniqueIntegerChromosome requires that no single gene be duplicated in the chromosome. These chromosomes are typically used when "Unique Integer" crossover methods are utilized (section 2.1.7.2).

The ChromosomeFactoryInterface is an Abstract Factory design pattern that is utilized by clients to the GAProblemInterface interface (see section 3.5.1.2). This interface specifies how chromosomes are created. Each subclass of ChromosomeBaseClass is expected to require the development of an implementation class of ChromosomeFactoryInterface. The implementation class would return an instance of the particular subclass of ChromsomeBaseClass when the method *CreateChromosome* is invoked. An implementation class for ChromosomeFactoryInterface is provided for all subclasses defined in the package. Many applications should be able to use one of the provided chromosome classes.

3.5.1.2 GA Utility Classes

Initially the intended design of this package was to require applications to derive themselves from a particular GA style (fitness proportionate, tournament, see section 3.5.1.6). While this worked successfully, redundant code still was required when it was desired to compare the two models on the same problem. There are two specific problems with this design. The first is the unnecessary code redundancy when doing comparisons between GA styles. This is undesirable as code duplication creates a software maintenance risk. Instead, the duplicated code should be delegated or encapsulated. The second problem involves the motivations for the inheritance. Only two minor reasons exist for the inheritance, to provide a fitness function to the GA and to monitor and log statistics about the GA. The solution to both problems realized the application of three design patterns.

An Observer pattern (section 3.3.3.1) was implemented to provide feedback detailing when a new best solution was found, what the periodic statistics were, when the GA had

finished initializing itself and when the GA was terminating itself. The **GABaseClass** provides the registration mechanics of the **GAObserverInterface** observer pattern. The GAObserverInterface defines callback methods for each one of the previously mentioned GA events. The class **GAObserverAdapter** (not an Adapter pattern) implements the GAObserverInterface. It provides default implementations for each of the abstract methods. A problem specific observer class can derive itself from the GAObserverAdapter class and only implement (override) the specific methods it is interested in. A package private implementation of the GAObserverInterface is the **GALocalReporter** class. It issues *Debug* and *DebugAndLog* calls to GADebugger (see further in this section).

A Strategy pattern (section 3.3.3.2) was designed using the **GAProblemInterface** interface. Instead of subclassing a particular GA style class, the user can instead implement the simple interface of GAProblemInterface. Here, two callbacks exist: one to test the fitness of a given chromosome, the other to create a chromosome factory (see section 3.5.1.1). This allows the GA style classes to be developed completely independently of user problems and even user defined chromosomes.

A significant class to the GA and GA derived packages is the **GADebugger** class. It provides a collection of static methods for generating debug messages. The class supports two forms of output media: Debug and Log. The *Debug* and *Debugln* methods are tied to an instance of a DebugWriter class (section 3.6.3). When the global property "DebugWriter" is enabled (per instructions of the DebugWriter class) a GUI window is shown allowing real time display of all "Debug" information. A second property, "gadebug.sys.out," can be defined independently of DebugWriter which also writes all "Debug" information to System.out. The methods Log and Logln write the given String object to a file in the current working directory called GADEBUG.LOG. The methods *DebugAndLog* and *DebugAndLogln* write the given String object to both the Debug stream and the Log stream as appropriate.

3.5.1.3 GA Resource Classes

The GA resource classes isolate and encapsulate the GA parameters for convenience and abstraction. A GA resource can be based on any source of IO that can implement the various interfaces. To start with, the **GABaseResourceInterface** presents the most basic and common parameters to the GA. Set and query methods are specified for each of the following parameters: best fitness target, mutation rate, maximum number of fitness evaluations, progress report rate, chromosome size, population size, crossover method, maximize or minimize, RNG seed, and various debug flags. GABaseResourceInterface implements the java.io.Serializable interface allowing instances of the interface to be transported via. a stream. **GABaseClassParameters** is the first class to implement the GABaseResourceInterface. It provides both the interface specified by GABaseResourceInterface and public variables to hold the actual parameters (as in a "C" language *struct*) that can be passed as a contained collection. GABaseClassParameters also implements the java.lang.Cloneable interface allowing easy duplication of the contents of the class. The class GABaseIni similarly implements GABaseResourceInterface but instead maps queries to reads from an encapsulated INIFile object. Sets were not implemented, as INIFile does not support writes. Extension of the resource interfaces is natural with respect to the difference selection methods. **TournamentGAResourceInterface** extends GABaseResourceInterface by adding methods to set and query the tournament size. **TournametGAParameters** extends GABaseClassParameters and implements TournamentGAResourceInterface adding an additional data member to hold the tournament size. **TournamentGAIniFile** extends GABaseIniFile and implements TournamentGAResourceInterface similarly. (This design can be extended into the Fitness Proportionate GA style. It was not for this thesis as only tournament selection was utilized)

3.5.1.4 Selection Method Classes

The selection classes are all derived from a common abstract base class: **GASelectionBaseClass**. Two specific subclasses are derived from GASelectionBaseClass: **RouletteWheelSelection** and **TournamentSelection**. These two classes are used by the FitnessProportionalGA and TouramentGA classes respectively in

the GA Style class collection. TournamentSelection provides a tournament selection mechanism characteristic of the Tournament style GA. RouletteWheelSelection provides the basic implementation of a fitness proportional style GA. Two subclasses to RouletteWheelSelection have been created: **RWSWithSigmaScaling** and **RWSByRank**. These classes implement the selection pressure modifiers discussed in section 2.1.6.1.

3.5.1.5 Crossover Method Classes

A single abstract base class exists for all crossover methods. **CrossoverBaseClass** allows crossover to be exercised without care to the specific crossover implementation. **SinglePointCrossover**, **DoublePointCrossover** and **UniformCrossover** are implementations of those crossover algorithms detailed in section 2.1.7.1. **CycleCrossover**, **OrderCrossover** and **PartialMatchedCrossover** are implementations of the crossover algorithms described in section 2.1.7.2. CrossoverBaseClass is one of the clients to the ChromosomeFactoryInterface as crossover algorithms need to be able to create child chromosomes to be placed back into the population.

3.5.1.6 GA Style Classes

There are two major GA style classes designed into this package: **FitnessProportionalGA** and **TournamentGA**. Both are derived from **GABaseClass**. The abstract class GABaseClass provides two sets of Strategy/Factory Method type plug-points which allow subclasses to override significant portions of functionality. The first Strategy pattern involves the initialization of the style class. The normal constructor invokes the *InitializeNow* method (which must be called sometime before the GA can be run even if it is not in the constructor). *InitializeNow* first calls *PreProcessGAResource* before extracting its own information from the associated GABaseResourceInterface instance. *PostProcessGAResource* is then invoked followed by *AllocateScores* and finally *AllocateChromosomes*. The *Pre/PostProcessGAResource* methods are useful since they allow subclasses to extract subclass specific information from the given GA resource. The two *Allocate* methods provide the hooks to subclasses to allocate their concepts of fitness scoring storage and chromosomes. An implementation of the fitness proportionate method may want to maintain two sets of scores: one for the current

population and one for the future population. An implementation of the tournament GA may need only one set of scores.

The second Strategy pattern involves the actual GA algorithm. The *run* method first invokes the *PreRunGA* method and then enumerates through the attached GAObservers invoking *GAInitializationComplete* on the interface. A *while* loop is executed conditioned on the value of *IsGADone*. The *while* loop invokes *PreGAIteration*, *GAIteration* and then *PostGAIteration* before returning to the conditional. *GAIteration* is where the main routine of the GA is expected to reside. When the conditional fails, *PostRunGA* is called followed by another enumeration of the attached GAObservers this time invoking the *GAFinalReport* method.

Maintenance of the GAObserver list is performed by GABaseClass. *InitializeNow* also initializes the list of GAObservers and attaches an instance of GALocalReporter. The public methods *AttachGAObserver* and *DetachGAObserver* both require GAObserver instances as parameters and either add or remove that instance from the observer list. Subclasses are not required to exercise (nor should they) any influence over the observer list. Instead, two important methods have already been implemented which simplify the operations of the GA style classes. Subclasses call *TestFitnessNow* to evaluate a chromosome (it takes a ChromosomeBaseClass instance as a parameter). It executes the GAProblemInterface method *TestFitness* on the associated problem instance utilizing the given chromosome and returns the obtained score. It also checks to see if the new score is better than any previously attained score (compared correctly depending upon a minimization or maximization problem). If so, it saves a reference to the chromosome and enumerates the GAObserver list invoking the *GANewHero* method. It also increments the fitness evaluation count. Incrementing the count can be enabled and disabled via the protected method *SetCountableEval* (this is utilized heavily by the parallel GA implementations). Subclasses can also invoke the *MakeProgressReport* method when desired (typically at the end of a "generation" of the GA). It first checks to see if the number of counted fitness evaluations has exceeded the next report threshold.

If-and-only-if so, the report threshold is incremented by the progress report rate and the GAObservers are enumerated with *GAProgressReport* being invoked.

FitnessProportionalGA subclass of GABaseClass is included as a placeholder within the package. It was originally intended to implement both GA style classes but time considerations prevailed as the TournamentGA implementation was solely utilized by the thesis. FitnessProportionalGA is responsible for implementing a generational replacement model using the fitness proportionate selection classes of section 3.5.1.4.

The GABaseClass subclass, TournamentGA, implements steady-state GA model with tournament selection (implemented in the TournamentSelection class). Its implementation of *GAIteration* contains an additional Strategy pattern. It invokes the following methods: *PreProcessSelection* (to instantiate the TournamentSelection object), *SelectFirstPair* and *SelectSecondPair* (to select the two winners and two losers), *CrossoverParents*, *MutateChildren*, *TestChildren* and *PostProcessSelection*. This strategy reflects the general GA steps of Figure 1. An invocation of *MakeProgressReport* is the last task of the method.

3.5.1.7 GA Package Summary

Applications using the GA package have very few requirements to get up and running:

1. Create an implementation of the GAProblemInterface.
2. Instantiate one of the GA style classes using an instance of the implementation in step 1.
3. Invoke the *run* method on the GA instance.

There are many ways to extend the implementations in this package. New chromosomes can be derived either from ChromosomeBaseClass or from any of the existing chromosome classes (with appropriate factory classes). New selection methods can be derived from GASelectionBaseClass or extended from existing classes. However, the appropriate GA style class will also need to be updated to allow the utilization of the new

class. Similarly, new crossover methods can be added but the GABaseClass will need updating to recognize the new classes. New GA style classes can be created by extending GABaseClass. It should be further noted that later analysis of the object design provided for a better abstraction of the selection classes and GA style classes. TournamentSelection should be utilizable by a generational GA model and RWS selection methods should similarly be accessible to a steady-state GA. Similarly, the GA style classes should not be tied directly to any single selection method. This abstraction flaw is carried into the CGGA packages. However, the CGGA packages are primarily concerned with the distribution of the GA itself, not how it works.

3.5.2 Basic CGGA package

The edu.rit.eecc.drp.cgga package extends the basic package by providing support for demes and parallel specific concepts.

3.5.2.1 Support Objects

The CGGA support object can be grouped into four further subcatagoties: parameters, chromosomes, GA Styles and other classes.

The interface **CGGAResourceInterface** does not extend the GABaseClassInterface. Instead it bases a new set of classes that parallel the GABaseClassInterface classes. CGGAResourceInterface defines methods for manipulating the migration rate, migration interval and deme synchronization parameters. The class **CGGAParameters** implements the CGGAResourceInterface paralleling the class GABaseParameters. The class **CGTournamentGAIniFile** extends TournamentGAIniFile and implements CGGAResourceInterface. This sufficiently extends the tournament GA specific INI file to include CGGA parameters.

The CGGA package requires some minor modifications to the chromosome classes of the GA package. The interface **CGGAChromosome** defines two additional methods that chromosomes in a CGGA environment must export: *GetId* and *SetId*. Chromosomes that are transferred between demes are part of the migration process. The

CGGAChromosome interface allows chromosomes to be uniquely identified throughout the global population. The class **CGGAUniqueIntegerChromosome** extends UniqueIntegerChromosome and implements the interface CGGAChromosome allowing the UniqueIntegerChromosome class to be utilized in the CGGA environment. **CGGAUniqueIntegerChromosomeFactory** extends UniqueIntegerChromosomeFactory and is responsible for defining and maintaining the uniqueness of the CGGAUniqueInterfaceChromosome instances throughout the global population.

Similarly to the chromosome classes, the GA style classes need some augmentation to execute properly in the CGGA package. The interface **CGGABaseInterface** defines some additional methods that GA style classes must implement for working with migration: *GetMigrants*, *SetMigrants* and *SetMigrationRate*. *GetMigrants* pulls from the GA an array of chromosomes dimensioned according to the size of the population and the current migration rate. Conversely, *SetMigrants* pushes an array of chromosomes into the GA. The GA is expected to replace some of its current population with the newly provided chromosomes. *SetMigrationRate* changes the migration rate of the GA. **CGTournamentGA** is derived from TouramentGA and implements CGGABaseInterface. It should be used when working with CGGA implementations.

Several "other" utility classes are included in the CGGA package. **CGGADebugger** extends GADebugger to allow for a CGGA specific debugger implementation (and to alleviate having to write *edu.rit.eecc.drp.ga.GADebugger.println("debug string")* throughout the code). **ProgressReportData** is derived from java.io.Serializable and encapsulates all of the data that a CGGADeme needs to report back to the CGGAMaster at the end of the migration interval.

3.5.2.2 Deme Objects

The abstract class **CGGADeme** is the starting point for using a CGGA to solve a problem. Subclasses need only to override the method *CreateCGGA* and optionally *AllocateThread*. *CreateCGGA* allows the subclass to instantiate an arbitrary

GABaseClass rooted object. However, CGGADeme checks that the returned object also implements CGGABaseInterface (use an instance of CGTournamentGA). CGGADeme utilizes the fact that GABaseClass implements the Runnable interface by placing the obtained GA into its own thread.

CGGADeme is derived from java.rmi.server.UnicastRemoteObject and acts as a server object within RMI. The *RunDeme* method is invoked by an executable class (e.g. the implementation subclass) availing the deme to be utilized in the network. CGGADeme also implements the **CGGADemeInterface** and the **CGGADemePeerInterface** interfaces. CGGADemeInterface and CGGADemePeerInterface are both subclasses of the java.rmi.Remote interface thus allowing CGGADeme to be accessed via RMI. CGGADemeInterface is the interface by which CGGAMaster objects access CGGADeme objects. CGGAMaster knows only of the CGGADemeInterface and not of the CGGADeme class or subclasses and therefore is only loosely coupled to the deme classes. CGGADemePeerInterface defines the interface between connected demes (peers). When the CGGAMaster instructs a deme that it is connected to another deme, the deme attaches to the peer via RMI on the CGGADemePeerInterface. The CGGAMaster does not know of this interface and therefore cannot affect it. Communication via this interface also allows different implementations of CGGADeme to exist on each node allowing for specialized applications of CGGAs. As CGGAMaster only knows of CGGADemeInterface, CGGADeme only knows of **CGGAMasterInterface**. CGGAMasterInterface defines the methods that a deme may invoke upon a master. This allows the master implementation to vary without influencing the CGGADeme. A master implementation need only implement CGGAMasterInterface to be recognized and utilized by CGGADeme.

3.5.2.3 Master Objects

The **CGGAMaster** class is the counter-part to the CGGADeme class. It too is derived from UnicastRemoteObject and runs as a server object within RMI. The *RunMaster* method is used by derived classes to release the master to its task. CGGAMaster

implements CGGAMasterInterface allowing CGGADeme to communicate with the CGGAMaster implementation without knowing what implementation is active. This becomes useful (and readily apparent) with the NUDGAMaster class.

During the execution of the master, progress reports are submitted from each deme. When a report is received, its contents are written to a log file separate from the CGGADebugger log. The GASTATS.LOG file is created in the same location as the master implementation class resides. Additionally, each time a deme is initialized, the *LogParams* method is invoked. This method writes the given GA and CGGA parameters to the GASTATS.LOG file.

CGGAMaster maintains a list of demes extracted from the contents of a NetworkTopologyDataFile (see section 3.6.3). The original implementation converted this list into a cache of remote CGGADemeInterface instances. An idiosyncrasy of the Win32 implementation of RMI prevents the caching scheme from working for more than eight or ten demes. The implementation was then changed to "lookup" the remote reference from RMI each time it is needed. While this is slower executing code, the scalability of the CGGAMaster is greatly extended.

The **NUDGAMaster** class is derived from the CGGAMaster classes. CGGADeme requires no modifications to work with NUDGAMaster. CGGADeme cannot differentiate between the two master classes as it interfaces exclusively with CGGAMasterInterface. The only changes NUDGA makes to the CGGAMaster is the overriding of methods *GetGAParameters* and *GetCGGAParameters*. These methods are overridden to implement the random assignment scheme for the GA and CGGA parameters, which is the point of NUDGA. A number of protected variables are available to subclasses of NUDGA that facilitate problem specific control and restriction of the created GA and CGGA parameters.

3.5.2.4 CGGA Package Summary

Applications intending to use the CGGA package have only a few simple steps to initially establish a successful environment:

1. Create a subclass of CGGADeme where the method *CreateCGGA* returns an instance of a CGGABaseInterface GA that works with an implementation of the GAProblemInterface.
2. Subclass CGGAMaster or NUDGAMaster and override its methods only if specialized behavior is desired.

3.5.3 GA2 Package

The edu.rit.eecc.drp.ga2 package extends the cgga package while incorporating an instance of the ga package.

3.5.3.1 Supervisory GA Classes

A single GA subclass is contained within the GA2 package. **GA2TournamentGA** extends the TournamentGA of the GA package. It overrides several of the Strategy methods that TournamentGA avails. The overriding does not alter the processing, instead the GA notifies the GA2Master of these significant events (via the **GA2InternalMasterInterface**, the methods are *SetChromosomeArray*, *InternalTesting*, *NewGA2Iteration* and *ChangedChromosomes*).

To support the GA2 GAs, the base class **GA2Chromosome** was created. Derived from BitChromosome, GA2Chromosome encodes all of the basic GA and CGGA parameters (crossover method, migration rate, migration interval, mutation rate, and population size). **GA2TournamentChromosome** extends GA2Chromosome further by adding the GA parameter tournament size. Factory classes exist for both.

3.5.3.2 GA2 Deme Classes

The **GA2Deme** class is a simple extension of the basic CGGADeme class and implements the interface **GA2DemeInterface.** GA2Deme overrides CGGADeme's *BindToRMI* and *UnbindToRMI* methods to allow it to also register with RMI as a GA2DemeInterface. This allows the GA2Master class to properly locate a valid

GA2Deme (not *just* a CGGADeme). This is essential to the operation of GA2. GA2DemeInterface defines the method *GA2DemeResetParameters* that is invoked by GA2Master when it decides to alter a GA2Deme's GA and CGGA parameters.

3.5.3.3 GA2 Master Classes

Unlike GA2Deme, the class **GA2Master** is a not-so-simple extension of NUDGAMaster. To facilitate the supervisory GA, GA2Master implements the GAObserverInterface and implements the GAProblemInterface. The GAObserverInteface allows the GA2Master to monitor the events and statistics of the supervisory GA. The GAProblemInterface specified the particular problem that the supervisory GA is to solve. That problem is the optimization of the progress of the GA2's demes. The best reported scores for each deme are averaged over a window of up to three migration intervals (obviously those demes just revised will not have three intervals worth of data and long standing demes will have more but are truncated to the latest three intervals). Note that GA2Master only utilizes the tournament style GA as coded in the class GA2TournamentGA.

3.5.3.4 GA2 Package Summary

Again, the instructions for utilizing the GA2 package are fundamentally the same as those for the CGGA package:

1. Subclass GA2Deme per instructions for CGGADeme.
2. Subclass GA2Master per instruction for NUDGAMaster.

3.5.4 Thesis Package

The edu.rit.eecc.drp.thesis package includes the implementation of specific code for working with the thesis problem. This package contains examples of how the ga, cgga and ga2 packages are expected to be utilized.

3.5.4.1 Thesis Utilities

Two significant utility classes developed for the thesis package are **ThesisCGGAParameters** and **ThesisProblem**. ThesisCGGAParameters extends CGGAParameters and encapsulates a TSPDataFile. This further generalizes ThesisDeme as it does not need to know what particular TSP instance is being solved. Instead, when

the CreateCGGA method is invoked, the CGGAParameter actually will contain an instance of ThesisCGGAParameters (*"isA"* CGGAParameter instance also) from which an instance of TSPDataFile can be extracted and utilized by ThesisDeme. ThesisProblem is an implementation of GAProblemInterface that tests chromosomes for their ability to solve an instance of the TSP. The constructor requires an instance of TSPDataFile that is utilized in the *TestFitness* method. *TestFitness* evaluates given chromosomes as tours through the TSP instance. *TestFitness* casts each chromosome to an instance of UniqueIntegerChromosome. In fact, *GetChromosomeFactory* returns an instance of CGGAUniqueIntegerChromosomeFactory that creates CGGAUniqueIntegerChromosomes that are derived from UniqueIntegerChromosome (section 3.5.2.1). ThesisProblem is utilized by the serial GA and the two deme implementations. The serial GA is unaware of the difference (it does not require the CGGA extensions) as it will utilize only the UniqueIntegerChromosome portion of the object (this is a virtual of object inheritance).

3.5.4.2 Thesis Serial GA

The **ThesisSerialGA** is an example of how simple it is to utilize the GA package. ThesisSerialGA's *main* method first instantiates a TSPDataFile, a ThesisProblem and a TournamentGA. The TournamentGA's *run* method is invoked and when that returns, all object references are nulled before the application is terminated. ThesisSerialGA extends the GAObserverAdpter class as it was necessary only to implement the GAProgressReport, GANewHero and GAFinalReport methods of the GAObserverInterface to maintain the specialized progress recording desired for this thesis.

3.5.4.3 Thesis Master Classes

Three "master" classes have been implemented, one for each CGGA master class: **ThesisMaster**, **ThesisNUDGAMaster** and **ThesisGA2Master**. Each class subclasses the associate master class and serves as an executable for initiating a run of the CGGA. Each class also overrides *DemeFinalReport* to delegate to **ThesisMasterSupporter's** *ProcessLastReport*. *ProcessLastReport* sends a report via SMTP to a preconfigured

postoffice and address if available. ThesisMaster also overrides *GetGAParameters* to alter the RNG performed by CGGAMaster. CGGAMaster does not alter the RNG seed thereby allowing each deme to be initialized to the same parameters. ThesisMaster maintains its own RNG seed (initialized to the original GAParameter value) and increments it by 137 for each subsequent invocation. ThesisNUDGAMaster similarly increments it by 1,777. ThesisGA2Master does not need to do this as GA2Master itself increments the RNG seed by 1,777. All thesis master classes override *GetCGGAParameters* to forcibly return an instance of ThesisCGGAParameters that contains an instance of TSPDataFile. ThesisDeme expects the CGGAParameters parameter to be an instance of ThesisCGGAParameters and extracts the TSPDataFile from it for later use by the deme. ThesisNUDGAMaster and ThesisGA2Master are required to adjust the crossover method picked by their parent classes. Since they are randomly chosen, it is possible to pick a crossover method that is not legal for ThesisProblem. ThesisMasterSupporter has an *AdjustCrossoverMethod* specifically for this purpose. ThesisMasterSupporter also has command line argument parsing methods and some additional methods to fetch the GA2 and NUDGA parameter limits from the limits file.

3.5.4.4 Thesis Deme Classes

The **ThesisDeme** class is a subclass of CGGADeme serving both the ThesisMaster and ThesisNUDGAMaster classes. **ThesisGA2Deme** is similar to ThesisDeme but is derived from GA2Deme. Both deme classes override the *CreateCGGA* method to allocate an instance of CGTournamentGA and to invoke **ThesisDemeSupporter**'s *CreateProblemInstance*. ThesisDemeSupporter is a delegation class for ThesisDeme and ThesisGA2Deme. It includes support routines that are common to both classes. Specifically, *CreateProblemInstance* returns an instance of ThesisProblem. Both classes also override *AllocateThread* and invoke ThesisDemeSupporter's *AllocateThread*. This delegation method returns a standard java.lang.Thread object with a priority set halfway between java.lang.Thread.NORM_PRIORITY and java.lang.Thread.MAX_PRIORITY.

3.6 Supporting Packages

The major packages described above required extensive use of several supporting packages.

3.6.1 awt

There are three classes in the edu.rit.eecc.drp.awt package which provide for a "better" set of base classes to inherit from for the purposes of creating windows and dialogs. **BasicWindow** is an abstract base class that adapts the normal **java.awt.Frame** to the event classes **java.awt.event.WindowListener** and **java.awt.event.ActionListener**. Java 1.1 introduced a revised event model by which GUI components interact with each other. The new event model uses delegation and modified Observer patterns (section 3.3.3.1). GUI components can register with other components by implementing known interfaces and attaching to the event sources. The WindowListener interface allows listeners to be notified of events concerning the actions of a window (open, close, minimize, maximize, activation, iconification). The ActionListener interface supports a single method: actionPerformed. *actionPerformed* has a ActionEvent parameter which indicates what type of AWTEvent has occurred and who the source of the event was. These two interfaces are usually necessary when developing a new window (Frame) class. The BasicModalDialog and BasicModallessDialog classes similarly implement the ActionListener and WindowListener interfaces but are instead derived from java.awt.Dialog not Frame.

3.6.2 lang

The edu.rit.eecc.drp.lang package provides a small collection of basic classes for utilization within the rest of the edu.rit.eecc.drp package. **ObjectPair** provides a simple and useful container for two objects (heterogeneous or homogeneous). Objects are classes ultimately derived from java.lang.Object, not built-in data types such as *int* and *double*. **Arrays** exports a number of methods to duplicate existing arrays and to print arrays. The two exception classes **NotYetImplemented** and **IllegalParameterException** provide convent exceptions for incremental development and for precondition checking. The exception **ValueNotFoundException** is a useful exception for search results.

3.6.3 io

This package (edu.rit.eecc.drp.io) augments the java.io package with some of the author's personal I/O routines. There are two types of classes included in this package: Java library extensions and specialized file I/O classes.

In the Java library extension category is the **FileFormatException** class. This class is utilized by the second category of classes in this package. The **Tokenizer** class provides a simpler means for tokenizing a stream than java.io.StreamTokenizer. It parses tokens from a java.io.Reader instance based upon user defined white-space characters. The class **DebugWriter** is an attempt to create a printable debug window that can be enabled or disabled via a global property "DebugWriter" (properties can be enabled with the "-D" flag as a command line parameter to the Java Runtime Environment (JRE)). Derived from the abstract class java.io.Writer, DebugWriter implements the necessary Writer interface. If the DebugWriter property is not enabled in the system, the debug window is not displayed and invocations on the interface to write data are ignored. If the DebugWriter property is enabled, the window is displayed and data is written to the window.

The second category includes classes for interfacing directly to three important file formats. **INIFile** provides an operating system neutral abstraction of the Microsoft Windows INI file format. Once instantiated, clients can read *boolean*, *double*, *float*, *int*, *long* and String values from the specified file (a *boolean* value are extracted as an *int* value and is converted to a boolean value via. bValue = nExtractedInt == 0). Although a write interface is technically allowable, none has been defined or implemented for this thesis. The INI file format is shown in Figure 22.

```
[SectionA]
EntryA1=ValueA1
EntryA2=ValueA2

[Section B]
EntryB1=ValueB1
EntryB2=ValueB2
;EntryB3=This entry is commented out
```

Figure 22: Sample INI File

The **NetworkTopologyDataFile** was developed to support the connection topology matrix necessary for the edu.rit.eecc.drp.cgga package. The intent of the file format is to allow a flexible means for describing the configuration of a network of devices.

```
<NetworkTopologyDataFile>
Name: some arbitrary name
Description: some arbitrary description
Dimension: some integer
Format: LIST
<addresses>
address1
address2
...
</addresses>
<listdata>
address1: address3
address2: address1 address3 address5
...
</listdata>
</NetworkTopologyDataFile>
```

Figure 13: Sample NTDF

Methods exist to extract address lists (the contents of the <addresses> section), get the "link to" addresses and query if an address is included in the address list. The format detailed must be adhered to exactly (white-space only where shown, lines in exactly shown order).

The **TSPDataFile** class provides an interface to the TSP data file format of TSPLIB95. Refer to section for more information about the TSP file format. Many methods exist to support access and utilization of the file (however, a complete interface to the file format of TSPLIB was not implemented for this thesis). To obtain the distance between any two cities the *Lookup* method is used. Since most of the distance calculations are complex and are time consuming, the TSPDataFile maintains a cache of all city pairs calculated. It has been determined that many of the city pair distances are repeatedly requested. This cache allows each unique city pair to be calculated only once saving many unnecessary (and costly) recalculations. TSPDataFile implements the java.io.Serializable interface allowing it to be written to a stream, cache and all.

```
NAME : att48
COMMENT : 48 capitals of the US (Padberg/Rinaldi)
TYPE : TSP
DIMENSION : 48
EDGE_WEIGHT_TYPE : ATT
NODE_COORD_SECTION
1 6734 1453
2 2233 10
3 5530 1424
4 401 841
5 3082 1644

...

44 7509 3239
45 10 2676
46 6807 2993
47 5185 3258
48 3023 1942
EOF
```

Figure 23: Sample TSP Data File (ATT48.TSP)

3.6.4 math

This package (edu.rit.eecc.drp.math) augments the math libraries of java.lang.math and the java.Math package. The **IntPair** class provides a container similar to ObjectPair of

edu.rit.eecc.drp.lang. However, this class wraps a pair of *int*s, Java primitive data types; strictly not subclasses of Object. The **Doubles** class initially started as a container for a method to check the equality of two *doubles*. It has since evolved to include inequality checking, string parsing to *double* values, and even a method to do simple rounding of a *double* value. The **Sort** class is a collection of several sorting algorithms: bubble sort, insertion sort, selection sort and shell sort. Several of the sort methods support alternative methods that return an array of integers indicating the sorted indexes. This is useful when it is not necessary to sorted the array but instead simply obtain a list of sorted indices. The **Vector** class provides a collection of methods for operating on vector arrays. Scalar multiplication, division, and addition as well as average, summation, and min/max search methods are all provided.

3.6.5 net

This package (edu.rit.eecc.drp.net) contains four additional classes to augments the java.net package. The class **IPSupport** provides two important methods: *GetLocalIPAddress* and *ParseIpAddress*. *GetLocalIPAddress* queries the local machine for its IP address. This method wraps several steps that are always executed when querying for this local address. *ParseIpAddress* was defined to assist tokenizes that are unable to ignore the decimal points found in IP addresses. Often, tokenizes will attempt to convert the IP address to a *double* and generate a java.lang.NumberFormatException because there are too many decimal points in the token. Instead, *ParseIpAddress* returns an ObjectPair instance where the first object is the extracted IP address token and the second object is the remainder of the unparsed line.

The remaining three classes support the communication protocol specified in RFC821 — Simple Mail Transport Protocol (SMTP). **SMTPClient** provides an interface directly into the SMTP protocol. Methods exist to resolve addresses and address lists and to send simple messages that can be contained in a single String object or to send larger messages as defined by **SMTPMessage**. SMTPMessage provides a simpler interface to the SMTP protocol by abstracting a message object. Clients work with SMTPMessage

objects to format messages that are more complex and then "hand" the object to an instance of SMTPClient to send. The **SendSMTPFile** class is an executable class that works with both SMTPMessage and SMTPClient to send a copy of a file using the SMTP protocol.

4. Data and Analysis

This section presents the various experiments of the thesis, the data collected, and the analysis of that data. The ATT48 problem from TSPLIB95 was selected for analysis. The ATT48 problem titled "48 capitals of the US (Padberg/Rinaldi)." TSPLIB has an optimal tour with a score of 10,624. Figure 24 shows this tour:

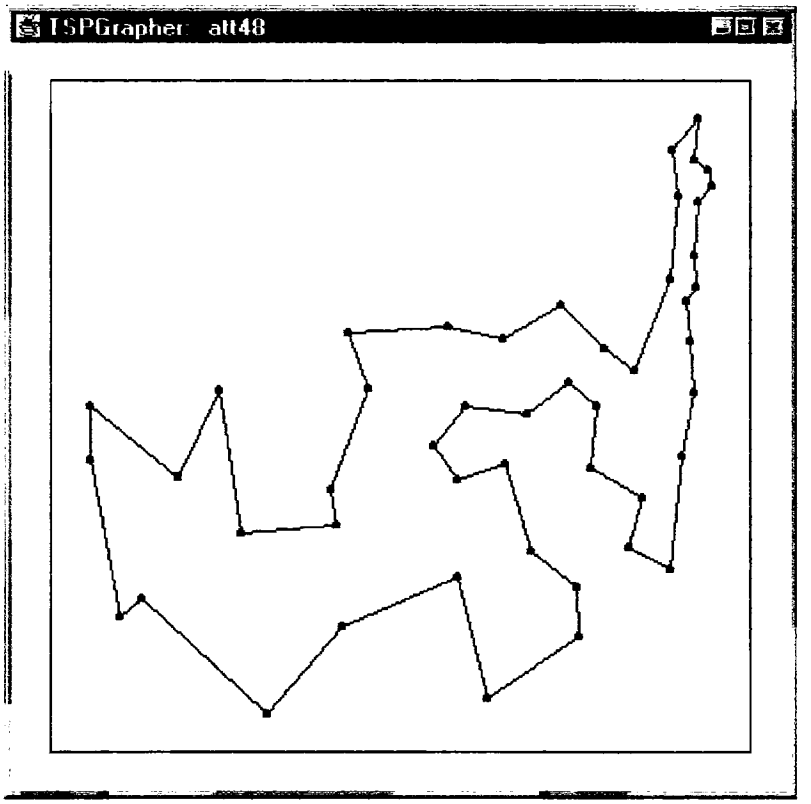

Figure 24: ATT48 with TSPLIB optimal tour

All runs of the three parallel genetic algorithms used a migration interval of 10,240 fitness evaluations and were connected using a ring topology. Five PCs were used for the runs, four machines for the deme nodes and one for the master node. The machines were not all evenly matched. Three of the deme nodes were Compaq Pentium II machines at 300MHz each with 96M RAM; the remaining deme node was a Gateway Pentium classic at 166MHz with 32M RAM (substantially slower). The master node was a Gateway Pentium Pro at 200MHz with 32M RAM. The network, too, was not optimal for the

trials. Two of the deme nodes were located on a different subnet of the local LAN from the other two deme nodes and the master node. Consequently, communication overhead cannot be fairly measured. The P5-166 simply cannot complete the same number of computations as the PII-300's can. Additionally, since two of the machines were not on the local subnet, additional communication delays were introduced by both the router and gateway between those on the subnet and those not. The delay is also subject to the available bandwidth of the network at the time the master or deme needs to communicate. These delays are unpredictable and subject to the randomness that is the modern computer networking environment.

All attempts to solve this problem involved a maximum of 1,024,000 fitness evaluations.

4.1 Basic Serial Genetic Algorithm

Twenty runs of the serial GA were performed with the following results:

	Serial
Best	10,909.0
Worst	13,955.0
Range	3,046.0
Average	12,626.2
Mean	12,657.0
Standard Deviation	801.4
% < 11000	5%
% < 12000	30%
% < 13000	70%

Table 1: Key Serial GA Statistics

All runs of the serial GA were done using partial matched crossover, 128 individuals, 10% mutation, and tournament sizes of 2.

This data shows that the serial GA had a moderate range of final solutions, while several runs broke the 11,000 barrier. The mean and median are closer to the "worst" score than the "best" score indicating that the most of the solutions were not approaching the 11,000

mark (which would indicate an approaching solution). Indeed, an inspection of Figure 26 and Table 1 shows that 70% of the tours failed to reach even the 12,000 mark.

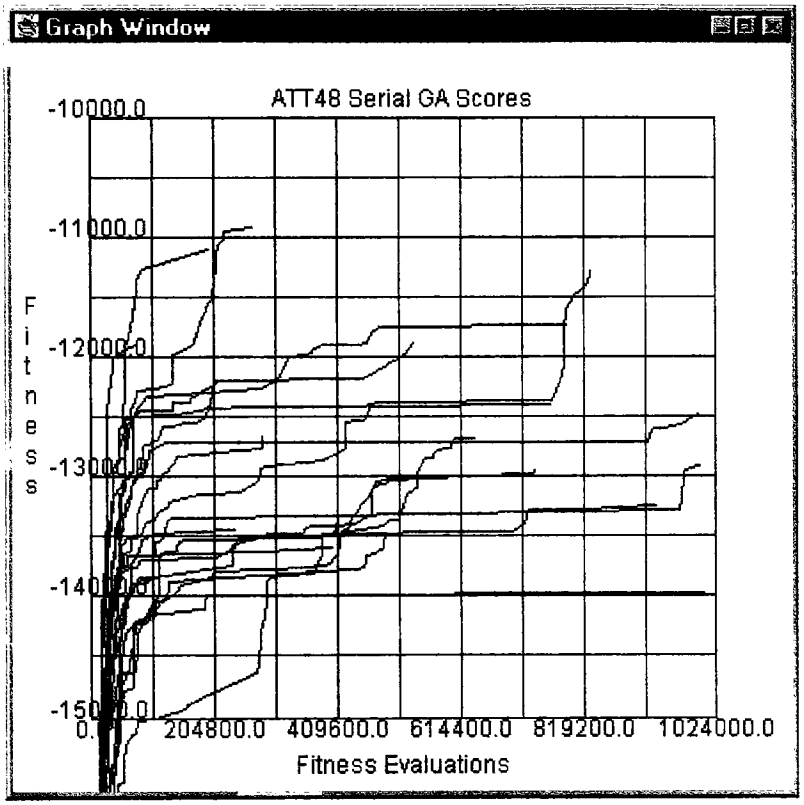

Figure 25: Serial GA Scores

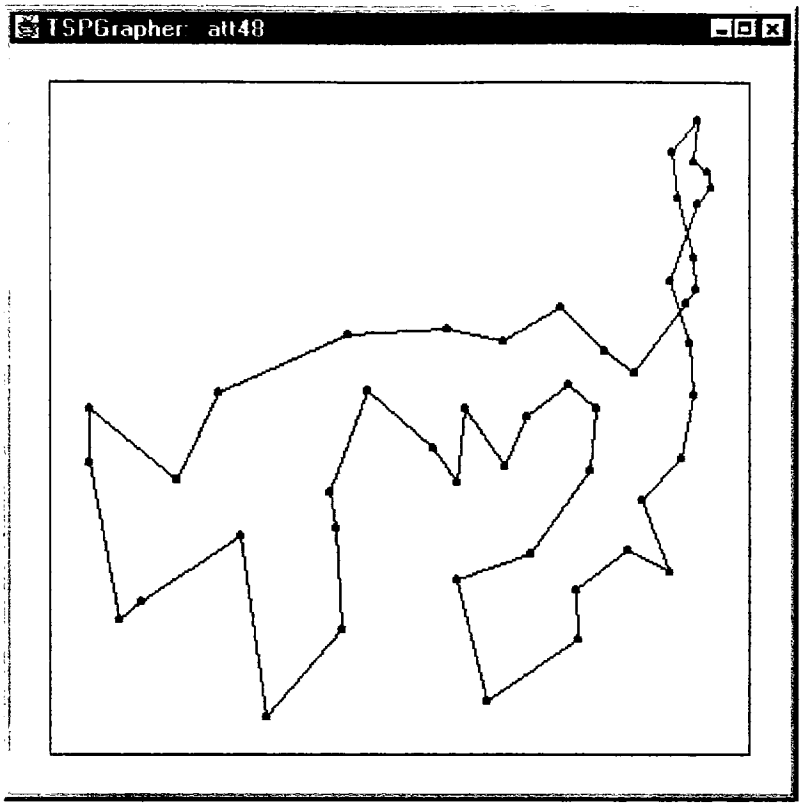

Figure 26: ATT48 Serial GA Seed 100109 Tour

The graph of the best tour realized by the serial GA is shown in Figure 26. The graph shows that a suboptimal tour was obtained. There are two edges that cross. The total tour length could be shortened simply by uncrossing those two edges. In total, the entire tour is approaching the form of the optimal tour (see Figure 24).

4.2 Basic Coarse-Grained Genetic Algorithm

Twenty runs of the CGGA were performed with the following results:

	CGGA
Best	11,151.0
Worst	13,706.0
Range	2,555.0
Mean	12,022.5
Median	11,806.0
Standard Deviation	781.8
% < 11000	0%
% < 12000	60%
% < 13000	90%

Table 2: Key CGGA Statistics

All runs of the CGGA were done using the same parameters as the serial GA: partial matched crossover, 128 individuals, 10% mutation, and a tournament size of 2. A migration rate of 10% was added to the rest of the serial GA's parameters. As is the intent with CGGA, all nodes are initialized with different RNG seeds.

It was unexpected that the CGGA did not find a better solution than the serial GA. However, while the CGGA did not better the serial GA tours it provided much more consistent scoring. The overall range of values decreased and the mean and median moved nearer to the "best" scores. The percentage of solutions less than 12,000 significantly improved over the serial GA.

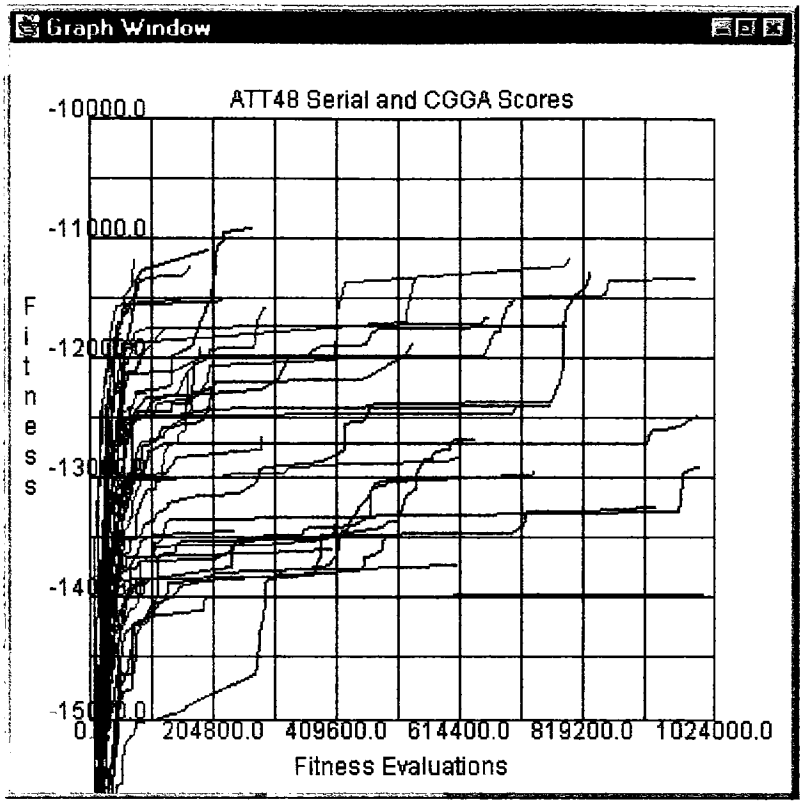

Figure 27: Serial and CGGA Scores

Note: Red lines are CGGA scores, and blue lines are serial GA scores.

4.3 Non-Uniform Distributed Genetic Algorithm

Twenty runs of the NUDGA were performed with the following results:

	NUDGA
Best	10,952.0
Worst	20,343.0
Range	9,391.0
Mean	14,179.1
Median	13,116.0
Standard Deviation	3,202.3
% < 11000	10%
% < 12000	30%
% < 13000	45%

Table 3: Key NUDGA Statistics

As might be expected, the NUDGA does perform worse than the CGGA. However, the data also shows that NUDGA can do better than the CGGA. The extreme range of solutions is due to poor parameters being chosen for the demes. It is obvious that better parameters can be chosen. The parameters of the best performing set were:

Parameter	Node 1	Node 2	Node 3	Node 4
Population Size	84	32	43	80
Tournament Size	5	5	5	2
Mutation Rate	18.30%	10.49%	34.99%	6.53%
Crossover Method	Order	Order	Cycle	Cycle
Migration Rate	2.03%	8.16%	24.82%	19.44%
RNG Seed	100043	101820	103597	105374

Table 4: Best NUDGA Parameters

It was surprising that partial matched crossover was *not* one of the crossover methods chosen this way. PMX significantly out-performed other methods during tests with the serial GA. In addition, tournament size is significantly more elitist than the serial and CGGA parameters. The average mutation rate is 17.577% that is 76% larger than the serial GA (and the CGGA). The average migration rate is 7.407% (26% less than the

CGGA). The population sizes selected here are also smaller than the serial GA. Preliminary tests with the serial GA using smaller population sizes did not perform well.

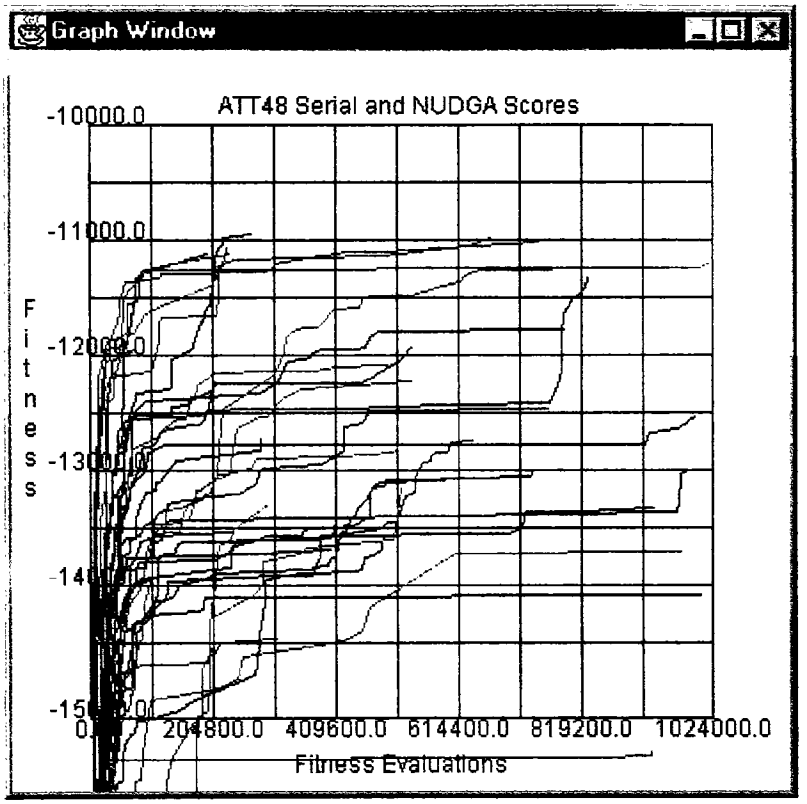

Figure 28: Serial and NUDGA Scores

Note: Magenta lines are NUDGA scores, and blue lines are serial GA scores.

4.4 Adaptive Distributed Genetic Algorithm

Twenty runs of the GA2 were performed with the following results:

	GA2
Best	10,777.0
Worst	11,355.0
Range	578.0
Mean	11,013.8
Median	11,010.5
Standard Deviation	141.3
% < 11000	45%
% < 12000	100%
% < 13000	100%

Table 5: Key GA2 Statistics

The GA2 results are noticeably better than all other implementations. The range of scores are more tightly focused with the standard deviation, median and mean appropriately reduced (compared to the other implementations). The best solution found was with RNG seed 100049 with a score of 10,777.

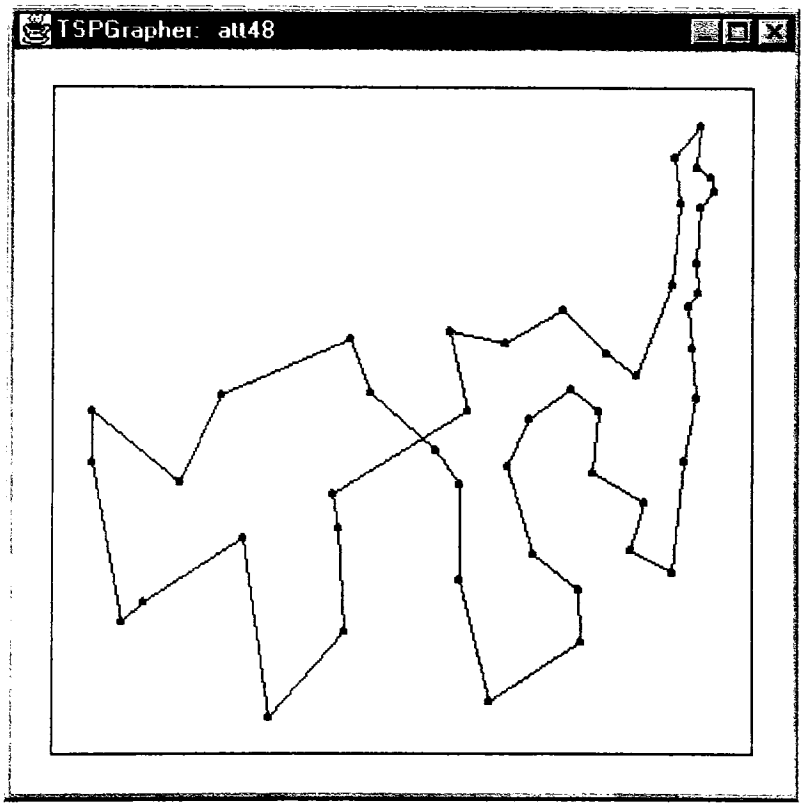

Figure 29: ATT48 GA2 Seed 100049 Tour

The following graphs show the variations of specific parameters over the course of RNG seed 100049. Each color represents one deme and the colors are consistent from graph to graph.

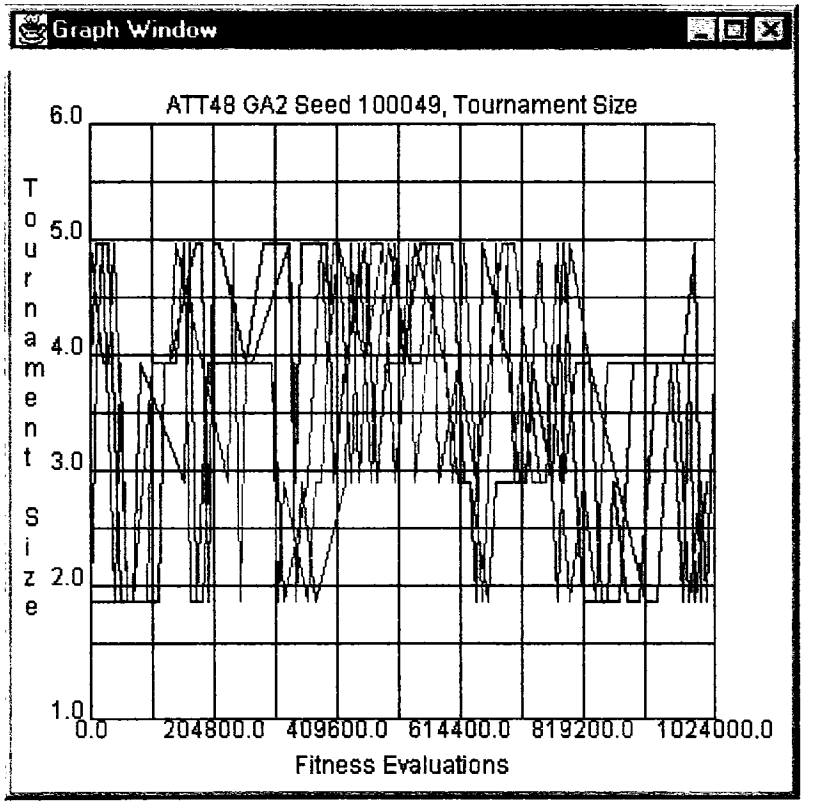

Figure 30: GA2 Tournament Sizes for seed 100049

Note: The graph's slight offset is due to rounding errors in the graphing software.

The tournament sizes are a combination of sizes and are not dominated by one specific size. This contrasts with the serial and CGGA in that both utilized a tournament size of two. In the beginning of the run there is a fair distribution of both small and large tournament sizes with a migration to sizes of strictly three, four, and five. There is a substantial number of changes in the middle region of the run reflecting that the supervisory GA cannot settle on a value until later in the run when it rejects the tournament size of five and reintroduces two.

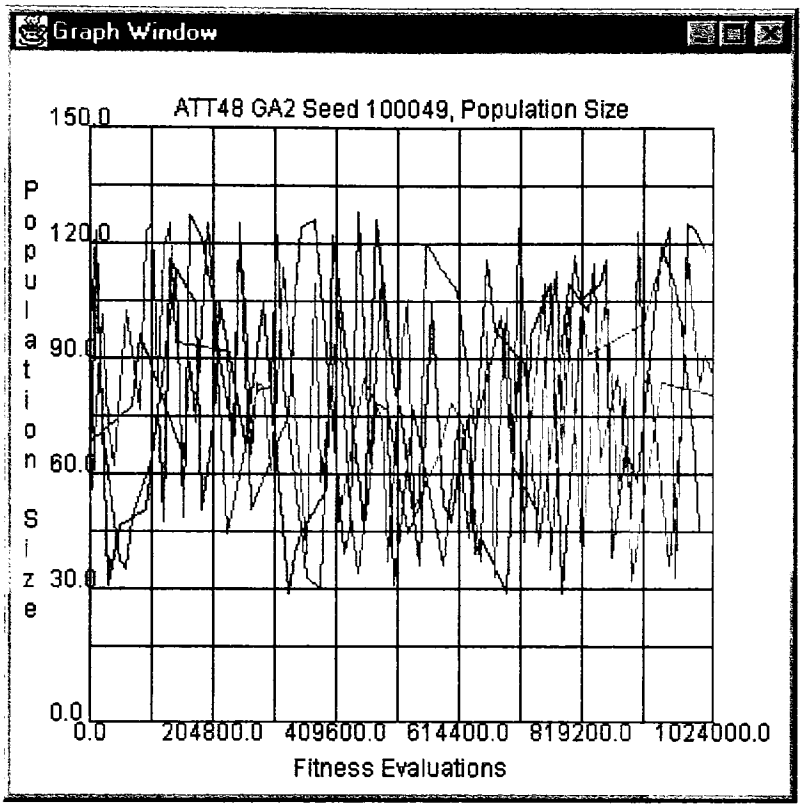

Figure 31: GA2 Population Sizes for seed 100049

As with the tournament size, there is no clear trend for population size. While there are many spikes indicating radical changes in size there is some consistency near where the average of all sizes would be.

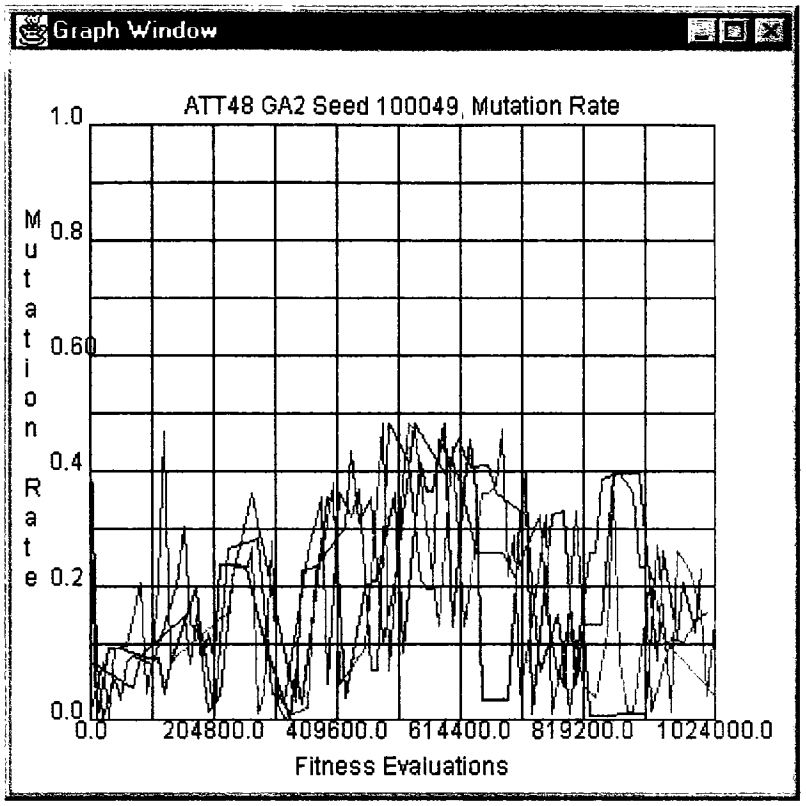

Figure 32: GA2 Mutation Rates for seed 100049

The mutation rates are clearly being adapted. The maximum allowable mutation rate was 0.5 explaining why the peak of the graph does not exceed this value. The last 400,000 evaluations appear to jump around with no clear trend. This is a function of the global population stagnating and the second level GA attempting to find a different set of parameters that will move beyond the current local maxima.

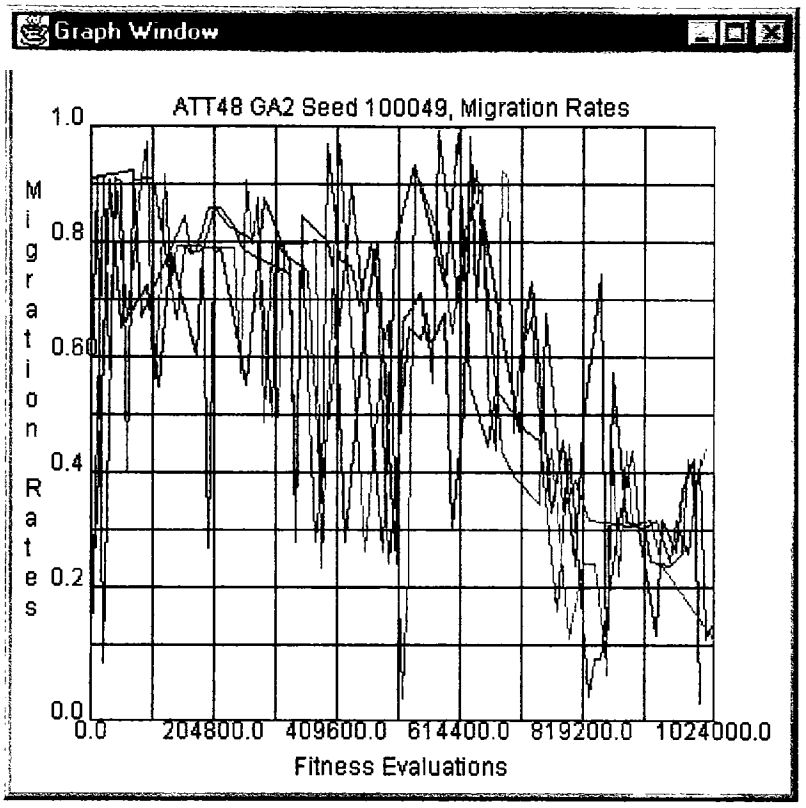

Figure 33: GA2 Migration Rates for seed 100049

The migration rate is also being adapted. It appears the migration rate is stable for the first 500,000 evaluations and is significantly reduced as the global population stagnates and the supervisory GA attempts to push past the local optimas. This graph is a primary indicator that adaptive mechanisms are necessary. The decrease in migration rate during the last half of the run is not completely due to randomness. Certainly, the initial few intervals did involve some randomness but the supervisory GA would not have continued to select deme parameters that performed poorly. Given that the other graphs are inconclusive in regard to a dominating value, it is clear that the migration rate can significantly affect the progress of the global population. It is also interesting to consider that the global population needs less migration later during the run. As the global population stagnates there is less need for migration because it is probable that each deme already has individuals similar to those being migrated. During the later stages, the

demes need new genetic material to further their progress through the fitness landscape. Certainly, a high migration rate is counter to this at this stage.

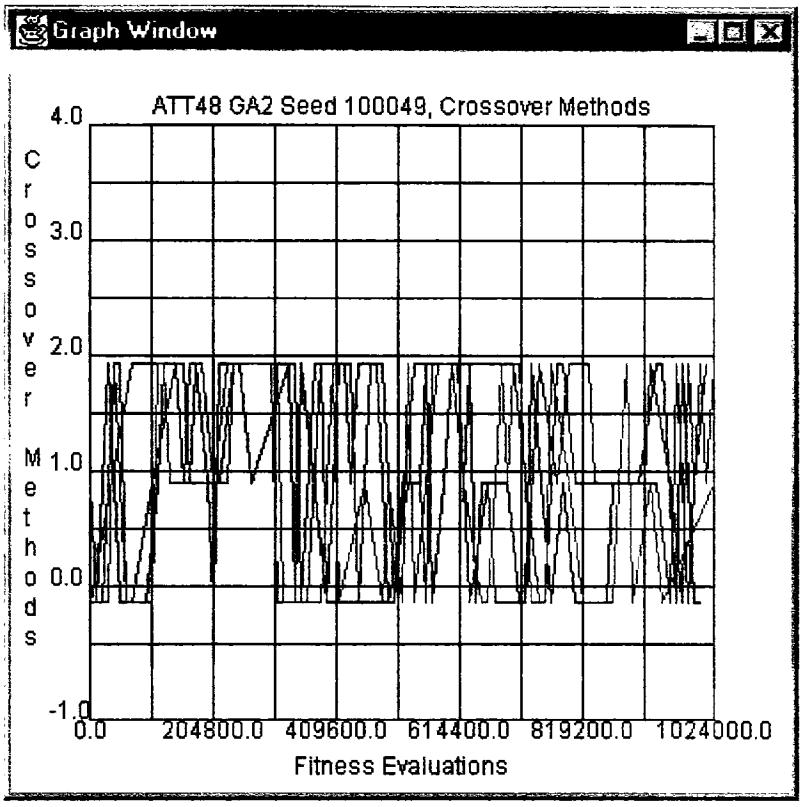

Figure 34: GA2 Crossover Methods for seed 100049

Note: The crossover methods have been adjusted such that 0 is cycle crossover, 1 is ordered crossover and 2 is PMX. Also, the graph's slight offset is due to rounding errors in the graphing software.

No crossover method dominates Figure 34. With the exception of cycle crossover, all methods are fairly represented during the coarse of the run.

These graphs are difficult to read. It should be expected that it is difficult to get a good idea of what the GA2 is doing with only four processing elements. 50% of the demes are being recreated at each synchronization. Larger collections of demes change a smaller percentage of all the demes allowing some demes to be unchanged for longer periods.

These longer stretches of time provide for a cleaner impression of the current state of the population.

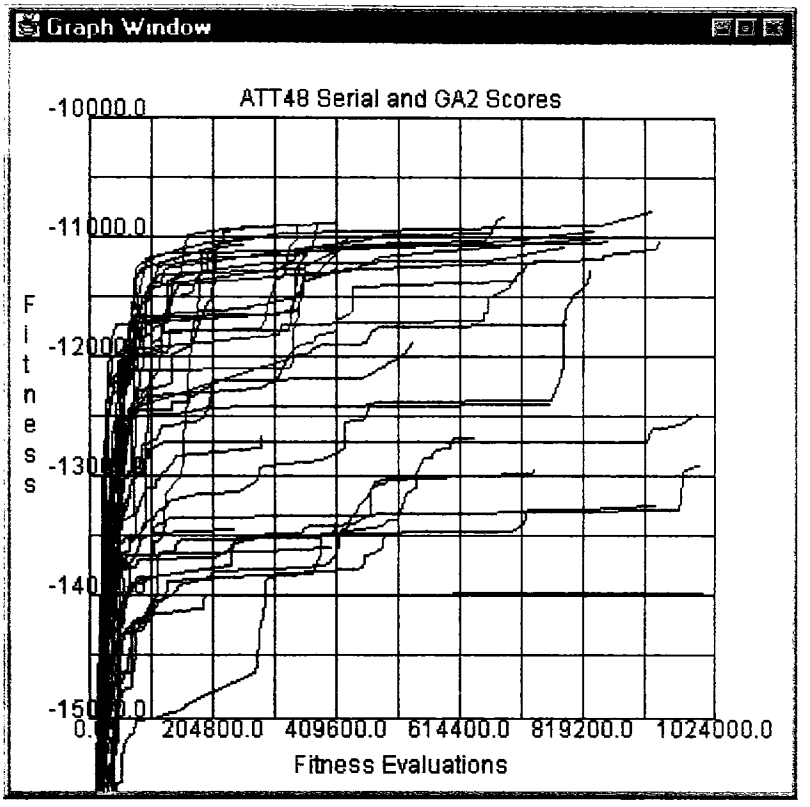

Figure 35: Serial and GA2 Scores

Note: Green lines are GA2 scores, and blue lines are serial GA scores.

4.5 Side-By-Side Comparison

Table 6 and Figure 36 present all of the statistics of the previous sections next to each other to facilitate easier comparing:

	Serial	CGGA	NUDGA	GA2
Best	10,909.0	11,151.0	10,952.0	10,777.0
Worst	13,955.0	13,706.0	20,343.0	11,355.0
Range	3,046.0	2,555.0	9,391.0	578.0
Mean	12,626.2	12,022.5	14,179.1	11,013.8
Median	12,657.0	11,806.0	13,116.0	11,010.5
Standard Deviation	801.4	781.8	3,202.3	141.3
% < 11000	5%	0%	10%	45%
% < 12000	30%	60%	30%	100%
% < 13000	70%	90%	45%	100%

Table 6: All GA Statistics Side-by-Side

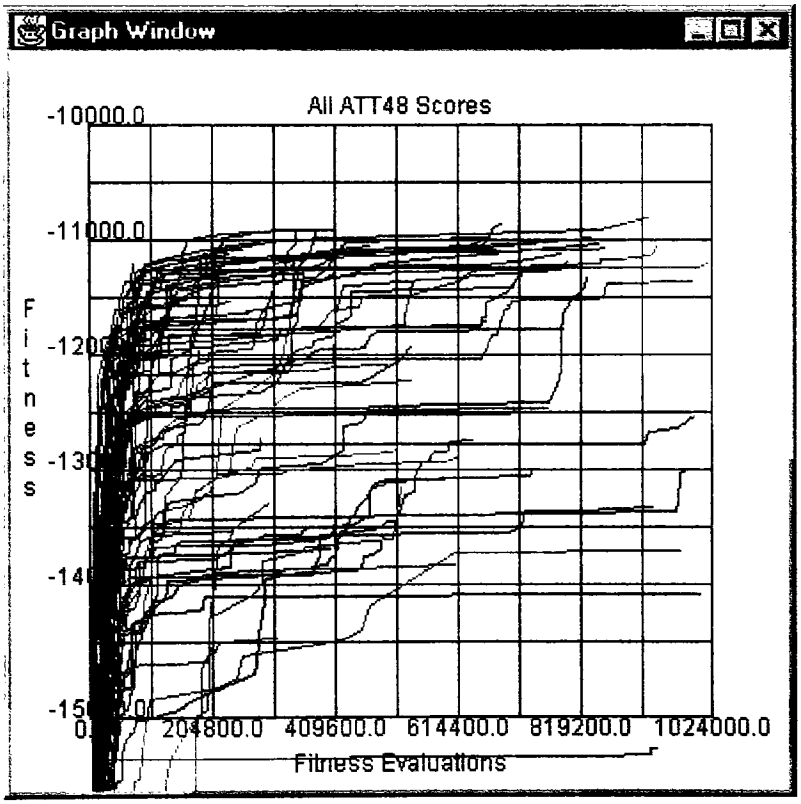

Figure 36: All GA Scores

Note: Red lines are CGGA scores, magenta lines are NUDGA scores, green lines are GA2 scores, and blue lines are serial GA scores.

It is clear from Figure 36 that while all of the GA implementations can do well the GA2 implementation is consistently better.

5. Conclusions

This thesis has focused on two issues: the development of a simple, flexible and extensible object-oriented GA and parallel GA library, and a comparison of the relative merits of three implements of CGGAs. Section 3.5 has presented a code library written in Java (version 1.1) that can easily be utilized to solve problems. Should additional research be desired, the library can readily be extended to accommodate new ideas.

Section 4 has clearly shown the benefits of the adaptive distributed GA. It is difficult to determine the best parameters for a GA without knowing a great deal about the problem being solved. Since the problem being solved is typically not understood it is contrary to have sufficient knowledge of the problem to establish the GA's parameters. Further, the optimal parameters may change during the execution of the GA. The typical GA, CGGA or NUDGA is not able to accommodate such evolution; the GA2 can. This makes the GA2 an ideal candidate for complex or poorly understood problems.

It is significant to note the marked improvement of the scores by adding only three additional processing elements to the serial GA. This is a minimal increase of total computing power and it should be expected that a small additional increase in processing elements would solve the problem.

5.1 Future Code Development

The following is a list of issues that would be beneficial or useful additions to the GA, CGGA or GA2 packages:

1. Add "go forever" option for unsynchronized tests.
2. Use the GA resource to gain dynamic loading of the crossover algorithm and the selection method at runtime. The crossover algorithm can be specified as an Uniform Resource Locator (URL) in the resource where it is currently a String object. Instead of converting the String value to a hard-coded class, the URL can be dynamically loaded. There are implications about how the NUDGA randomly selects class and how the GA2 chromosome works. The dynamic specification of the selection

method is somewhat harder and would require some rework. A Factory method in combination with the URL specification should be sufficient.

3. Make selection method an element of the NUDGA initialization and GA2 chromosome implementations.
4. Add a greedy crossover option (i.e. accept children only if they are better than the parents).
5. Add MPX to the list of crossover methods.
6. Optimize the cycle, ordered and partial matched crossover algorithms. Specifically, replace the L^2 algorithms with less computationally costly ones. This is the most significant problem related to solving larger TSPs (i.e. problems with more than 500 or 1,000 cities).
7. Develop *int* (primitive data type, not java.lang.Integer) variants of the permutation crossover methods in which compares are less costly.
8. Add the ability to stop the GA and save the state so that it can be restarted from where it left off.
9. Add dynamic restructuring of the network.

5.2 Future Research Areas

The success of the GA2 and DAGA2 architectures shows that adaptive mechanisms are necessary for, and useful to, the generic genetic algorithm. Further, the success of the iiGA implementation shows that hybrids hold significant potential for solving problems. It should be expected that further research into both will yield increasingly better designs and solutions. The following is a list of research topics to be considered in the future:

1. Investigate the use of a larger number of deme nodes. This will increase the depth of the genetic pool allowing for the determination of either a quicker solution or a more extensive search of the problem.
2. Investigate the use of higher orders of connectivity between demes.
3. Implement a hybrid deme that implements a TSP "branch-and-cut" algorithm.
4. Investigate unsynchronized networks.

6. Bibliography

[BD93] Bertoni, A. and Dorigo, M. (1993). Implicit Parallelism in Genetic Algorithms. Artificial Intelligence, 67, pp. 307-314.

[CP971] Cantú-Paz, E. (1997). *Designing Efficient Master-Slave Parallel Genetic Algorithms*. (IlleGAL Report No. 97004), Urbana, IL: University of Illinois at UrbanaChampaign.

[CPE95] Cantú-Paz, E. (1995). *A Summary of Research on Parallel Genetic Algorithms*. (IlleGAL Report No. 95007). Urbana, IL: University of Illinois at UrbanaChampaign.

[CPE97] Cantú-Paz, E. (1997). *A Summary of Research on Parallel Genetic Algorithms*. (IlleGAL Report No. 97003 - revision version of IlleGAL Report No. 95007). Urbana, IL: University of Illinois at UrbanaChampaign.

[CW94] Corcoran, A.L. and Wainwright, R.L. (1994). A Parallel Island Model Genetic Algorithm for the Multiprocessor Scheduling Problem. *Proceedings of ACM/SIGAPP Symposium on Applied Computing*. (pp. 483-487). ACM Press.

[DJS90] De Jong, K., and Spears, W.M. (1990) An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms. *Parallel Problem Solving from Nature*.

[ED97] Eby, D., et. all. (1997). *An Injection Island GA for Flywheel Design Optimization*. (GARAGe Technique Report 97-05-04). Michigan State University Genetic Algorithms Research and Applications Group/Case Center.

[FD96] Flanagan, D. (1996). Java in a Nutshell. Sebastopol, CA: O'Reilly & Associates, Inc.

[GD89] Goldberg, D.E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. New York: Addison-Wesley.

[GDC92] Goldberg, D.E., Deb, K. and Clark, J.H. (1992). Genetic Algorithms, Noise, and the Sizing of Populations. Complex Systems, 6, pp. 333-362.

[GEA95] Gamma, E. et. all. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley Publishing Company, Inc.

[HCGM] Harik, G., Cantú-Paz, E., Goldberg, D.E., and Miller, B.L. (1997). The Gambler's Ruin Problem, Genetic Algorithms, and the Sizing of Populations. In Bäck, T. (Ed.) *Proceedings of the Fourth International Conference on Evolutionary Computation.* (pp. 7-12). New York: IEEE Press.

[JF95] Jones, T., and Forrest, S. (1995) Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms, In Eshelman, L.J. (Ed.) *Proceedings of the Sixth International Conference on Genetic Algorithms.* (pp. 184-192). San Francisco, CA: Morgan Kaufmann Publishers.

[LPG94] Lin, S., Punch, W.F. and Goodman, E.D. (1994). Coarse-Grain Parallel Genetic Algorithms: Categorization and New Approach. In *Sixth IEEE Symposium on Parallel and Distributed Processing.* (pp. 28-37). Los Alamitos, CA: IEEE Computer Society Press.

[MG95] Miller, B.L. and Goldberg, D.E. (1995). *Genetic Algorithms, Tournament Selection, and the Effects of Noise.* (IlleGAL Report No. 95006). Urbana, IL: University of Illinois at UrbanaChampaign.

[MH90] Mühlenbein, H. (1990). Parallel Genetic Algorithm in Combinatorial Optimization. In Rawlins, G. (Ed.) *Foundations of Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann Publishing.

[MH91] Mühlenbein, H. (1991). Evolution in Time and Space - The Parallel Genetic Algorithm. In Rawlins, G. (Ed.) *Foundations of Genetic Algorithms.* (pp. 316-337), San Mateo, CA: Morgan Kaufmann Publishing.

[MM96] Mitchell, M. (1996). An Introduction to Genetic Algorithms. Cambridge, MA: The MIT Press.

[MS95] Mahfoud, S.W. (1995). Population Size and Genetic Drift in Fitness Sharing. In Whitley, L.D. and Vose, M.D. (Eds.) *Foundations of Genetic Algorithms 3.* (pp. 185-223). San Francisco, CA: Morgan Kaufmann Publishing.

[MW92] Mathias, K. and Whitley, D. (1992). Genetic Operators, the Fitness Landscape and the Traveling Salesman Problem. In Mannor, R. and Manderick, B. (Eds.) *Parallel Problem Solving From Nature 2.* (pp. 219-228). North Holland-Elsevier.

[OGC91] Oei, C.K., Goldberg, D.E. and Chang, S. (1991). *Tournament Selection, Niching, and the Preservation of Diversity.* (IlleGAL Report No. 91011). Urbana, IL: University of Illinois at UrbanaChampaign.

[RFCs] http://www.ohio-state.edu/htbin/rfc/rfc-index.html

[SDJ96] Sarma, J, and De Jong, K. (1996). An Analysis of the Effects of Neighborhood Size and Shape on Local Selection Algorithms. In *Parallel Problem Solving from Nature IV*. (pp. 236-244). Berlin: Springer-Verlag.

[SG97] Semeraro, G. (1997). Evolution of Solutions to Real-Time Problems. Rochester, NY: Rochester Institute of Technology Master's Thesis.

[SFP93] Smith, R.E., Forrest, S. and Perelson, A.S. (1993). Searching for Diverse, Cooperative Populations with Genetic Algorithms, Evolutionary Computing, 1, (pp. 127-149).

[SM95] Schwehm, M. (1995). Massively Parallel Genetic Algorithms. In Eshelman, L.J. (Ed.) *Proceedings of the Sixth International Conference on Genetic Algorithms*. (pp. 520-527). San Francisco, CA: Morgan Kaufmann Publishers.

[SM97a] Sun Microsystems. (1997). AWT Enhancements. In JDK 1.1.4. file:E:\java\docs\guide/awt/index.html

[SM97b] Sun Microsystems. (1997). RMI Specification. In JDK 1.1.4. file:E:\java\docs\guide\rmi\spec/rmiTOC.doc.html

[TSPLIB] Reinelt, G. (1995). Universität Heidelberg. http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html.

[WDP96] Wang, G., Dexter, T.W., and Punch, W.F. (1996) Optimization of a GA and within a GA for a 2-Dimensional Layout Problem. In Goodman, E., Punch, W., and Uskov, V. (Eds.) *Proceedings of the First International Conference on Evolutionary Computation and its Applications*. Russian Academy of Sciences.

[WGP96] Wang, G., Goodman, E.D. and Punch, W.F. (1996). *Simultaneous Multi-Level Evolution*. (GARAGe Technique Report 96-03-01). Michigan State University Genetic Algorithms Research and Applications Group/Case Center.

[WGP97] Wang, G., Goodman, E., and Punch, W. (1997). *On the Optimization of a Class of Blackbox Optimization Algorithms*. (GARAGe Technical Report 97-02-01). Michigan State University Genetic Algorithms Research and Applications Group/Case Center.


CD-R
COMPACT DISC RECORDABLE
TITLE
RICOH
TYPE
74
650MB
DATA
Coarse-Grained Parallel Genetic Algorithms
by Daniel R. Pedersen
Rochester Institute of Technology
Department of Computer Engineering
© 1998
COMPACT
disc
Recordable